Modulating Application Behaviour for Closely Coupled Intrusion Detection

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Marc Welz
June 2003

Supervisor: Dr A. C. M. Hutchison

Abstract

This thesis presents a security measure that is closely coupled to applications. This distinguishes it from conventional security measures which tend to operate at the infrastructure level (network, operating system or virtual machine). Such lower level mechanisms exhibit a number of limitations—amongst others they are poorly suited to the monitoring of applications which operate on encrypted data or the enforcement of security policies involving abstractions introduced by applications.

In order to address these problems, the thesis proposes *externalising* the security related analysis functions performed by applications. These otherwise remain hidden in applications and so are likely to be underdeveloped, inflexible or insular. It is argued that these deficiencies have resulted in an over-reliance on infrastructure security components.

The thesis introduces a *design* to externalise this functionality, realised as an *interface* between applications and an external security component which acts both as *intrusion detection system* and *voluntary reference monitor*. The interface can be seen to offer applications a means to solicit a second opinion when performing security related tasks. Compared to conventional intrusion detection systems, this *direct coupling* makes the analysis less vulnerable to desynchronisation, furnishes it with higher quality information and makes fine-grained, preventive measures feasible.

A nontrivial *system* based on this design has been *implemented*, *released* and *deployed*. Selected results describing its effectiveness, performance impact and the effort required to operate the system are reported. These confirm that the proposed approach is viable in operational systems and suggest that closely coupled intrusion detection and response can be achieved by modulating application behaviour.

Acknowledgements

I would like like to thank my supervisor, Dr Andrew Hutchison who, despite other commitments, remained supportive and approachable. I am also grateful to the MSc students of the Data Network Architectures laboratory: Farrel, Lourens, Mwelwa, Ben, Simon and the others who arrived or left during my studies—thank you for your company.

Since I worked outside an established security research group, I was largely dependent on electronic resources. Articles and papers that had been made available online without distribution restrictions were particularly helpful. I am thankful for the comments of the users of my implementation, especially the detailed bug reports submitted by Andras Bali. Andras also packaged the IDS/A implementation for the Debian GNU/Linux distribution [13]. Further bug reports and a patch implementing `idsa_vsyslog` were submitted by Matthew Kirkwood.

I am indebted to Thomas Andrews and Dr Richard Lord who were kind enough to go over the draft and made valuable suggestions. Thanks also to Regine Heuschneider, Paul Sheer and my brother Eric for proof-reading the thesis. I am grateful to the National Research Foundation which funded my studies. Finally and most importantly, I would like to thank my parents for their unwavering support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Summary

Despite considerable research [114, 31, 18, 124, 30, 64, 21], it remains remarkably difficult and expensive to construct secure software. This has particularly pronounced implications for applications, as these tend to be developed under significant resource constraints. Consequently a large number of flaws are discovered in applications only after they have been deployed, making it necessary to limit and otherwise manage the impact of security flaws at runtime.

Usually the first line of defense against the exploitation of such flaws are infrastructure access controls, whether provided by the host operating system, virtual machine or network firewall. While these are important barriers, this thesis contends that they are not sufficient to protect *trusted* applications that are required to interact with potential attackers. This restriction has contributed to the introduction of other security mechanisms such as audit modules and intrusion detection systems. In particular, intrusion detection systems provide substantial analysis facilities which can be used to monitor applications for undesired activity.

Although attractive, conventional intrusion detection systems are also not without limitations—they are, as noted by Ptacek and Newsham [104], vulnerable to *desynchronisation*. Desynchronisation occurs when an intrusion detection system fails to track the state of the monitored system and its environment accurately. Such failures may result in attacks going unnoticed or legitimate activity being interpreted as threatening. The latter makes it difficult to deploy automated countermeasures as attackers may be able to trigger the intrusion response to disrupt legitimate activity.[1]

[1]This problem has an analog in certain types of autoimmune failures encountered in

This thesis argues that desynchronisation opportunities arise when the guarding systems function at a significant distance from the guarded ones. In this context the distance is not only spatial or temporal[2] but also conceptual: Intrusion detection systems tend to operate at lower levels of abstraction and thus are required to perform an expensive translation process. Effectively this demands the duplication of significant application (and occasionally even infrastructure) functionality to a high degree of fidelity.

This thesis proposes a stronger integration between detection system and guarded application, as the reduced separation presents fewer possibilities for attackers to decouple the security mechanisms and so evade detection. To explore this approach a nontrivial system has been designed, implemented and published. The implementation provides a bi-directional and dedicated interface which allows application developers to transfer security analysis tasks to an external system, permitting not only more accurate monitoring but also fine-grained responses which block undesired activity.

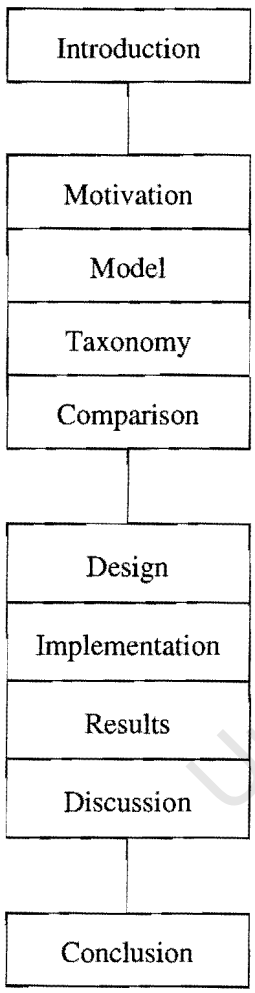
This instrumentation has been added to several existing and new trusted applications and results indicate that this approach can be useful to enhance application security with moderate costs.

biological systems.

[2]Although both can also be used to desynchronise a monitor.

1.2 Thesis Structure

The thesis can be divided into two logical parts—an analysis (Chapters 2 - 5) and a synthesis (Chapter 6 and following). The analysis consists of the following four chapters:



Chapter 2 defines the topic of application security and motivates why it is worth investigating.

Chapter 3 introduces a general model that describes the security components protecting applications and maps several common approaches to it.

Chapter 4 uses the model as the basis for a taxonomy which identifies five major attributes describing the security components.

Chapter 5 reviews the security component design classes and examines the tradeoffs that characterise the classification space induced by the taxonomy.

The analysis provides the context for the introduction of the security component described in the second part. It contains the following chapters:

Chapter 6 describes the proposed security component and orientates it in the classification space of the analysis.

Chapter 7 introduces a concrete and released implementation of the design.

Chapter 8 reports results which relate to the implementation effort needed to instrument applications, the performance impact and the security gains.

Chapter 9 discusses the limitations of the approach as well as social factors and possible extensions.

Chapter 2

Motivation

2.1 Introduction

This chapter argues that an investigation of application security is both relevant and challenging. It notes that application security failures are numerous and expensive, while resource constraints make it difficult to construct perfectly secure applications. This makes it necessary to manage the flaws of deployed applications.

2.2 Definitions

As there exist several interpretations of the terms *application* and *security*, this chapter defines both.

2.2.1 Applications

This thesis defines the term *application* as *"a process or program executing outside the context of its host runtime"*.

A runtime can take various forms but usually includes a component which manages the execution (operating system, virtual machine or script interpreter) and a collection of predefined supporting functions (libraries).

In other words, an application is considered to be any execution context on the calling side of the application programmer interfaces (APIs). This definition is a broad interpretation—unlike the narrow one which is restricted to end user programs such as word processors or mail clients, this definition also includes server processes and system utilities such as database managers or backup tools.

The computer systems in common use tend to provide reasonably well-defined boundaries between host infrastructure and application in the form of published APIs, typically implemented as system calls, library interfaces or object invocations.

This does not imply that the boundary between application and infrastructure is fixed—it shifts when additional components offering new APIs are made available. In such cases a component previously part of an application may become sufficiently useful and a prerequisite for other applications to be considered part of the host infrastructure. In other words, the boundary between application and host infrastructure is not necessarily static but changes as the capabilities of the infrastructure are expanded. In some cases this may result in nested applications—for example, a web browser may simultaneously be considered an application at one level (a process hosted by an operating system) and runtime at another (an interpreter for a scripted mortgage calculator).

The distinction between application and host runtime, whether static or not, is useful as the interface between the two represents the boundary between general and specialised systems in a form which is used to build the majority of computer systems.

From a security perspective it is interesting to understand the degree to which general purpose infrastructure can be used to secure applications and the circumstances under which it is necessary to rely on applications to operate securely.

Such an investigation could be considered topical as current attempts of large information producers to exert greater control over the use of information appear to generate a renewed interest in trusted operating systems and infrastructure in general.

2.2.2 Computer Security

There exist two common interpretations of the term *security*—a broad one that includes most failures or undesirable conditions and a restricted one that requires the presence of an attacker either to exploit a failure or otherwise extract an advantage.

Examples of the former, broad interpretation include the one given by Olovsson [97] which states that *"A secure system is a system on which enough trust can be put to use it together with sensitive information"*, or another found in the NSA glossary maintained by Stocksdale [129] which defines computer security as *"technological and managerial procedures applied to computer systems to ensure the availability, integrity and confidentiality of information managed by the computer system."*

An example of the latter, narrow interpretation is the informal definition as given by Cheswick [32] which states that *"...security is keeping anyone from doing things you do not want them to do to, with or from your computers or any peripherals"*. A more formal version is provided by Howard [62]: *"Computer security is preventing attackers from achieving objectives through unauthorised access or unauthorised use of computers and networks."* A similar approach is taken by Meadows [91] who defines security as including *"...any means for ensuring that a computer-based system performs a function in the face of an intruder or intruders who are actively trying to prevent it from doing so"*.

Although the broad definition of security may be more pervasive, this thesis will use the second, narrow definition which requires the participation of an adversary and will use the terms *reliability* or *safety* to refer to the broad interpretation[1]. In other words, insecure systems are considered a subset of unreliable systems.

2.3 Relevance

This thesis asserts that the topic of application security is worthy of investigation, as application security failures account for a significant fraction of all published and exploited vulnerabilities.

For example, of the 748 Common Vulnerabilities and Exposures (CVE) dictionary entries assigned for the year 2000 (CVE-2000-0001 to CVE-2000-1189, as listed in [35]), application failures outnumber other failures by a factor of five.[2]

In addition to being numerous, application failures have the potential to be expensive. Data supporting this claim can be found in a publication of Computer Economics [41] which contains an estimate of the cost of major computer security incidents. The three most expensive incidents are reproduced in Table 2.1.

All three incidents involve applications: Both the Love Bug and Sircam take advantage of design flaws in Microsoft Outlook (a mail client), while Code Red propagates by exploiting a vulnerability (CVE-2001-0500) in a

[1]Consider the spontaneous mechanical failure of a harddisk. Under a strict interpretation of the broad definition this would have to be considered a security problem as it it impacts negatively on integrity or availability.

[2]The analysis conducted as part of this thesis classifies 40 entries as kernel failures, while 73 relate to embedded systems or other difficult to classify systems (*eg* a flaw in VMware, a virtual machine, can be interpreted both as an application failure as well as a hardware flaw). As there exist a number of cases where the classification is ambiguous, only an approximate and conservative ratio of 5:1 is given.

Incident	Year	Economic Impact (USD)
Love Bug	2000	8.75 Billion
Code Red	2001	2.62 Billion
Sircam	2001	1.15 Billion

Table 2.1: Cost Estimate of Major Security Incidents [41]

component of Microsoft IIS (a web server). The thesis notes that quantifying the cost of a computer security incident is difficult, and that the above figures may overstate the impact. However, the relative ranking is more certain—there seems little reason to doubt that the above incidents are among the most expensive ones reported thus far.

The large number of application failures as well as their potential severity makes efforts that aim to secure applications relevant. Given the scale of the problem and lack of progress in improving security (as noted by Blakley [23] and others), alternative and even marginal approaches are worthy of investigation.

2.4 Challenges

Even when substantial resources are allocated to security it remains difficult to build nontrivial, yet secure computer systems. This problem is well-known, having been noted some time ago by Saltzer and Schroeder [114] who stated that *"even in systems designed and implemented with security as an important objective, design and implementation flaws provide paths that circumvent the intended access constraints"*. Recent examples confirming this observation include failures in OpenSSH and OpenBSD [38] (see CVE-2002-0083, CVE-2002-0639 and CVE-2002-0640), systems which have previously been subjected to extensive security analysis.

Compounding this problem is the fact that in many cases security is only a secondary goal as current economic and legislative conditions do not sufficiently reward those who attempt to build secure software. Security competes with other requirements (such as time to market, feature set, ease of use) for scarce resources. Usually the most efficient tradeoff from the perspective of those producing software is to allocate only limited resources to security.

Anderson [6] notes that this behaviour is rational: Reaching the market before a competitor or having more features are both likely to improve revenue, while the cost of security failures is borne by the user, not the vendor,

as vendors of computer related products are currently able to disclaim all responsibility for failures of their systems. This is in contrast to providers of other products and services where those harmed by failures tend to have legal recourse. Similarly, individuals involved in the production of software are not required to be licensed as is the case in other engineering fields.

This thesis does not take a position on the desirability of changing these conditions by, for example, requiring that software engineers be professionally registered or regulating markets to force vendors to absorb the costs of computer failures. It merely observes that currently there exist only limited incentives to produce secure software and that security competes with other requirements for limited resources—consequently expensive techniques (formal proofs, comprehensive testing) are unlikely to be deployed in the near term.[3]

The effect of this lack of resources is particularly pronounced in the case of applications as, relative to the likes of operating systems or network protocols, applications tend to:

- have shorter life-cycles,
- be more numerous,
- be more varied and
- be developed by less qualified or experienced developers.

Under these conditions several techniques known to reduce the number of security flaws are either less likely to be applied or become less effective. Amongst others:

- A shorter development cycle leaves less time for testing, while a shorter deployment period reduces the chance that flaws found during operational use will be repaired.
- Costly and time-consuming formal verification methods are even less likely to be applied to applications than to operating systems or network protocols.
- As applications are more diverse it is more difficult to establish a set of well-known and proven designs and an associated body of knowledge.

[3]However, there exists the possibility that the greater security needs of vendor lock-in modules and digital restrictions management systems [6] may increase incentives.

Compared to infrastructure components, applications experience less pressure to remain static. Infrastructure components are depended on by other systems and are expected to retain the same interfaces across revisions—this limits the rate at which changes can be introduced. Applications tend to be less constrained in this regard, hence application development is more dynamic, proceeding at a greater pace and yielding more numerous and diverse systems.

A greater rate of change increases the potential for the introduction of flaws. Thus even under the assumption that the producers of applications are not malicious and may even make some attempts to secure applications, it is reasonable to assert that applications containing security flaws will continue to be shipped for some time to come.[4] This makes defences that guard against the exploitation of such flaws particularly relevant.

2.5 Approach

The previous sections established that is is difficult to build secure systems, that in general few resources are allocated to security objectives and that these problems are particularly pronounced in the case of applications.

Under such conditions it becomes important to deploy the few resources that are available in a way that maximises their effectiveness. A well-established strategy used to pursue this goal calls for system designs that concentrate security functions at particular modules or components. One of the earlier suggestions to this effect can be found in a report by Anderson [4]; the report recommended that a *security kernel* be built, where *"the objective of a security kernel design is to integrate in one part of an operating system all security related functions."* If effective, such designs make it possible to focus resources allocated to security on a subset of the system. Flaws in the remainder of the system, while still being a reliability concern, have reduced or even no security implications.

In order to achieve this objective, means have to be found to modularise the security responsibilities of a given system to such a degree that they can be transferred to dedicated and conceptually distinct *security components*. This thesis uses the term *security component* in preference to *security kernel* to indicate that this strategy has led to the development of several systems, not all of which are strongly integrated with operating systems. Virus scan-

[4] Although some progress has been made in limiting the impact of applications flaws with the deployment of Java Virtual Machines [80], Managed Code Infrastructures and related sandbox environments, Chapter 3 will argue that a class of applications can not be protected completely by such mechanisms.

ners, firewalls, intrusion detection systems, audit components and operating system reference monitors are all examples of such security components—either compensating for security deficiencies in, or assuming security responsibilities of, other systems including applications.

Despite the existence of such security components, application failures remain numerous and expensive. Thus there may be some merit in investigating the extent to which it is feasible to transfer the security responsibilities of applications to these systems. An improved understanding of the circumstances under which application security responsibilities can (or, perhaps more importantly, cannot) be transferred to other systems should be of practical value to both producers and users of applications. Such knowledge could also be applied in the design and implementation of alternative security components.

This thesis conducts such an analysis by using a simple security component model to establish a framework within which several common security system designs can be classified and their characteristics described. The framework also provides the context for the analysis of security component weaknesses and the examination of newer approaches aimed at overcoming these limitations.

Chapter 3

Model

3.1 Introduction

This chapter presents a model which describes systems that are designed to replace or reinforce the security functions of other entities. These systems are referred to as *security components*. Although the model can be applied more generally, this thesis focuses specifically on the security components that are used to protect applications containing accidentally introduced flaws.

3.2 Model Description

The operation of a security component guarding another system can be reduced to three functions:

1. *Collecting* information relating to the guarded system
2. *Analysing* the information gathered to establish whether the guarded system satisfies given security requirements
3. Mounting a *response* dependent on whether the requirements are being met

A diagram of the security component model is provided in Figure 3.1: Data about the guarded system is acquired and mapped by the decoding module to a set of security events. These serve as input for the analysis module which establishes whether a security event constitutes a security failure by referencing a security policy or equivalent (encoded as an access control list, set of firewall rules or database of IDS signatures). This decision is communicated to the response module which reacts accordingly.

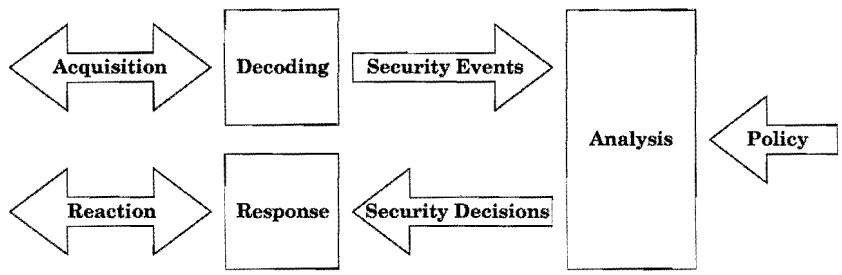


Figure 3.1: A Simple Security Component Model

Similar models have been applied previously in several security related contexts:

- The Common Intrusion Detection Framework (CIDF) project developed an architecture described in Porras *et al* [101] as part of an effort to define an interoperability mechanism among different intrusion detection systems [128]. This architecture defines four component types: Event generators (*"E-boxes"* in the terminology of the CIDF project), event analysers (*"A-boxes"*), event databases (*"D-boxes"*) and response units (*"R-boxes"*). The *E*, *A* and *R-boxes* of the CIDF architecture correspond, to a degree, to the respective *decoding*, *analysis* and *response* subcomponents of the model used in this thesis.

- An analysis of logging and auditing by Bishop [19] describes the process of monitoring a system as consisting of a *logging* and an *audit* step, where the *audit* step is further divided into *reduction*, *analysis* and *notification* steps. The *logging*, *analysis* and *notification* steps of the security logging model can be thought of as being analogous to the respective *decoding*, *analysis* and *response* subcomponents of the model introduced here, while the function of the *reduction* step could be distributed over the *decoding* and *analysis* subcomponents.

The model proposed in this chapter differs from that of the CIDF project in that it is used as an abstract analytical framework, thus the model entities are not necessarily interchangeable or even distinct at the implementation level, as would be the case with CIDF elements.

3.3 Mapping

This section maps several well-known security component designs used in the protection applications to the above model. It also examines some of the tradeoffs made by the respective components.

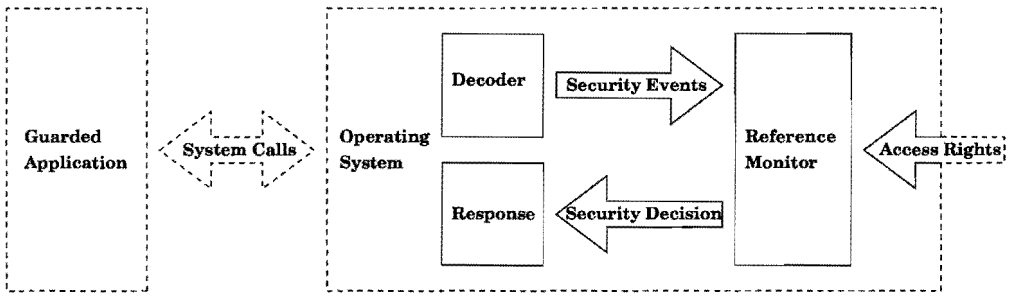


Figure 3.2: Conventional Host Reference Monitor mapped to the Security Component Model

3.3.1 Host Reference Monitors

The access control subsystems provided by the infrastructure of an application are the primary security barrier of most computer systems. These subsystems, also known as *reference monitors*, regulate access to the resources provided by the infrastructure. For example, most operating systems invoke a reference monitor component to establish if an application may read a particular file, while at the network level a firewall may be used to regulate the traffic reaching a networked application.

With respect to the protection of flawed applications, infrastructure reference monitors can be used to restrict the access of potential attackers to applications, as well as to limit the damage which can be caused when attackers do subvert applications.

In terms of the model, host reference monitors are security components tightly integrated with the infrastructure. A request by an application for a resource is received by the infrastructure, decoded to a security event (usually in the form of a subject-object-access triple) and submitted to the analysis module. In most cases the analyser performs no more than a table lookup or equivalent to establish whether the request should be granted. This decision is relayed to the response module which either makes the resource available to the application or reports an error. A diagram illustrating how a reference monitor can be mapped to the security component model is given in Figure 3.2.

Host reference monitors are well-established security components—their utility has been described by Anderson [4][1] and most computer systems in current use include reference monitors that implement access controls based

[1]The report uses the term *Reference Validation Mechanism* when referring to implementations of the reference monitor concept. This thesis will use the term *Reference Monitor* for both the concept and its implementations.

on designs set out by Lampson [76]. However, although these are generally valuable and widely deployed, there do exist certain conditions under which these security components are of limited utility.

Consider the example of a flawed mail transfer agent. Amongst other privileges, this application is permitted by the host infrastructure to read incoming mail messages from the mail port. An attacker, interested in subverting the mail transfer agent, writes a corrupting mail message to the mail port. The message triggers a flaw in the mail transfer agent—a consequence of insufficient length checking, a quoting deficiency or another oversight. This allows the attacker to gain control of the application and, by proxy, to acquire the remaining capabilities of the mail transfer agent, such as the ability to manipulate mailboxes. The noteworthy aspect of this type of attack is that it may occur even when lower level access controls are in force.

It is possible to examine the limitations of conventional infrastructure reference monitors in terms of the access matrix model described by Lampson [76]. An access control matrix is used to map a subject-object pair to a set of access rights. Subjects are active entities seeking access to resources, objects are the resources protected by the reference monitor, and access rights are the operations that the reference monitor will perform on objects at the request of subjects.

The subjects of interest in this thesis are attackers and applications. Attackers are subjects aiming to increase their capabilities beyond those explicitly granted in the access matrix, while applications are subjects containing flaws which, when exercised, allow an attacker to gain control of the application.

This problem is independent of the issue of access control *safety* (addressed by the likes of Sandhu [116]), which seeks to understand when an attacker can gain additional access rights through legitimate updates of the access control matrix. Instead this investigation focuses on how a static access matrix might be used to prevent an attacker from subverting another subject.

The application flaws considered here are introduced accidentally, not maliciously, making it improbable that they can be triggered via a covert channel. Given this assumption, an attacker has to interact with an application using a communication channel that can be regulated by a host reference monitor. In other words, an attacker has to be in possession of capabilities which permit some interaction with a flawed application before it can be exploited, where applications of interest are those that possess capabilities not already available to the attacker.[2]

[2]In this context denial of service attacks can be thought of as attempts to acquire the

The above properties can be described in terms of the capability sets of attacker and application, where the term *capability set*[3] refers to the set of access rights and object pairs that are available to a given subject.

The notation developed as part of this thesis defines the capability set of a subject (C_s) as a set of ordered pairs, each pair consisting of an object (o) and an access right set (a), where a is the set of operations that s may perform on o. Given a Lampson [76] access matrix (A), it is possible to derive the capability set for a given subject as follows:

$$C_s = \{(o, a) \mid a = a_{s,o}\} \tag{3.1}$$

Where $a_{s,o}$ is the set of access rights at row s and column o of matrix A, each row referring to a subject and each column to an object.

For an attacker to be interested in subverting a given application the application should possess capabilities not already available to the attacker. This can be expressed as follows:

$$\exists (o, a_{v,o}) \in C_v, (o, a_{t,o}) \in C_a : a_{v,o} - a_{t,o} \neq \emptyset \tag{3.2}$$

Here C_t and C_v are the capability sets of the attacker and the vulnerable application respectively.

In order for an attacker to exploit a vulnerable application, there has to exist some means of communicating with the application. Such a mechanism can be modelled using a set of objects which can transmit and receive information, acting as message containers. The transmissions are subject to access control, inducing two access right subsets: a^w, the set of rights enabling a subject to write information to an object and a^r, the set of rights allowing a subject to read information from an object. Thus for an attacker to send a message to an application, there has to exist an object to which the attacker can write and from which the application can read. This can be expressed as follows:

$$\exists (o, a_{v,o}) \in C_v, (o, a_{t,o}) \in C_t : a_{v,o} \cap a^r \neq \emptyset, a_{t,o} \cap a^w \neq \emptyset \tag{3.3}$$

Similarly, a message can be sent in the opposite direction, from application to an attacker, if the following is true:

$$\exists (o, a_{v,o}) \in C_v, (o, a_{t,o}) \in C_a : a_{v,o} \cap a^w \neq \emptyset, a_{t,o} \cap a^r \neq \emptyset \tag{3.4}$$

capability to suspend or terminate an application.

[3]The terms *protection domain* [94] or *information domain* [118] are sometimes used for similar purposes.

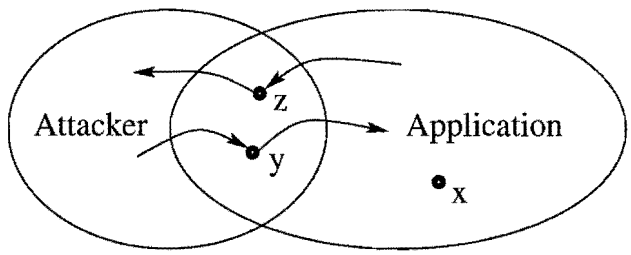


Figure 3.3: Interaction between Attacker and Vulnerable Application

Usually access controls are used to block this communication entirely, while multilevel security systems are designed to restrict this flow of information to one direction only. The classical MLS approach focuses on confidentiality and blocks writes to lower levels (excluding 3.4), while integrity-centric designs prohibit reads from lower levels (disallowing 3.3).

This thesis contends that blocking such exchanges is often too restrictive. In particular, a large number of real world applications, ranging from electronic commerce suites to multi-user gaming environments, are intended to communicate with other systems—this is the appeal of a networked environment. Thus these applications satisfy one, and usually both,[4] of the above statements.

However, if subjects communicate but have different capabilities, a failure in one subject may allow another to gain access to the capabilities of the failed subject, even if the capabilities are guarded by infrastructure access controls. A diagram illustrating this is given in Figure 3.3, where objects x, y and z can be bound to o in expressions 3.2, 3.3 and 3.4 respectively.

This thesis will refer to applications that satisfy expression 3.2 and (either 3.3 or 3.4) as being *trusted* and to the remainder as being *trusting*, noting that the security responsibilities of a trusted application cannot be transferred completely to conventional infrastructure reference monitors. In contrast, a trusting application can to a reasonable degree rely on the infrastructure to either isolate it from hostile subjects or restrict it to capabilities that are already in possession of the subjects interacting with it. Either measure diminishes the security implications of a flaw in a trusting application significantly, reducing the security concerns to ensuring that such trusting applications do not contain malicious flaws and are not deployed in roles where only trusted application would suffice.

Consider the example of a trusting desktop calculator application: Host access controls are sufficient to restrict this application to the same privileges

[4]Most widely-used protocols are bidirectional because duplex communication is desirable to correct errors and essential for interactive systems.

as the user invoking it and to ensure that only one user has access to a given calculator instance. There may well exist flaws in the calculator, however, its user has no incentive to exercise them, as this would yield no new capabilities and would only adversely influence the user's own computations. In other words, the flaws of this application are largely an issue of reliability rather than security, as it is possible for a reference monitor to assume most security functions of the calculator. This changes as soon as the calculator is used in a hostile environment where an unreliable system becomes a direct security hazard. For example, if the parts of the calculator are reused in an electronic commerce system (a trusted application), a flaw might allow an attacker to purchase an item at an unauthorised discount.

This thesis notes that increased efforts aimed at partitioning or isolating subsystems are likely to reduce the possibilities that a trusting application may accidentally be exposed to hostile subjects, while finer-grained infrastructure access controls are an effective means of limiting the capabilities which an attacker may gain by compromising a trusted application. However, the thesis conjectures that neither fine-grained host access controls nor isolation mechanisms are sufficient to eliminate trusted applications entirely.[5]

An example supporting this notion is a trusted medical database application which permits a certain user to retrieve a particular disease incidence rate for a population but not the disease status of an individual. The disease incidence rate is a domain-specific abstraction introduced by the application and has its own security requirements.

Controlling access to such domain abstractions at the infrastructure level is nontrivial, as the mapping from infrastructure resources (disk blocks) to abstractions (disease averages) can be arbitrarily complex, requiring either the duplication or inclusion of substantial application functionality in the infrastructure reference monitor. Consider the above database example where the calculation of the population average may generate the same disk block reads as a listing of all individuals—distinguishing between the two requires access to the database logic.[6]

Conventional operating system reference monitors are generally not used to perform this task, as it is at odds with the principles of Anderson [4]. These state that reference monitors should be tamper-proof, always invoked and small enough to be well tested. In particular, the provision of facilities to encode and apply the domain knowledge necessary to follow higher level exchanges between attacker and application may conflict with the simplicity

[5]This position can be rephrased using the terminology of Clarke and Wilson to state that it does not appear feasible to transfer all application resident transformation procedures to conventional infrastructure.

[6]Blocking inference attacks at the infrastructure level poses similar problems.

and tamper-resistance requirements.

In terms of the security component model, the decoding module of a conventional operating system reference monitor generates security events which relate to the access of infrastructure resources. In order to maximise reliability, the analysis of these security events is relatively simple, usually consisting of stateless lookups to determine if an access request should be granted.

Reference monitors of this type are sufficient to isolate flawed but nonmalicious[7] applications from other subjects, but provide insufficient protection for trusted applications that are *required* to interact with potential attackers. This limitation has led to the introduction of other security component designs such as intrusion detection and audit systems which are used to discover failures not prevented by conventional reference monitors.

3.3.2 Network Intrusion Detection Systems

Network intrusion detection systems (NIDS), of which NSM [60] is an early representative, are entities that monitor network traffic for signs of undesirable activity. Network intrusion detection systems can be thought of as standalone security components which, amongst others, guard networked applications such as chat clients, P2P systems or mail servers. Along with infrastructure access controls and virus scanners, NIDS are among the most frequently deployed security components.

A conventional NIDS can be mapped to the security component model as follows: The data source of a NIDS is network traffic which is collected using a promiscuous network interface or network tap. Each network packet is considered a security event which is submitted for analysis. NIDS analysis modules occur in two major forms:

Misuse detectors apply a set of rules or signatures which describe known illegal activity

Anomaly detectors employ statistical or machine learning techniques to flag deviations from normal activity

Both NIDS analysis forms tend to be more sophisticated than the equivalent infrastructure reference monitor subsystems, as the latter generally perform no more than simple table lookups. The same is not true of NIDS response modules, which are usually less developed and not as effective as

[7]The confinement of applications intent on violating such constraints is substantially more difficult [77] and is arguably not feasible using conventional infrastructure.

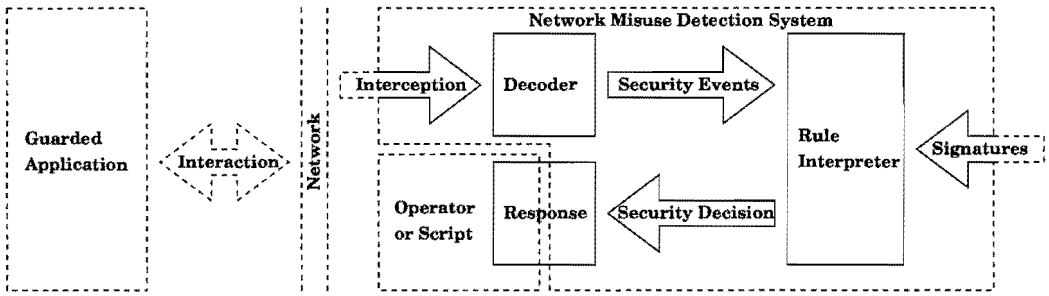


Figure 3.4: Conventional Network Misuse Detection System mapped to the Security Component Model

the host reference monitor equivalents, as the possibility exists that a NIDS may intervene in error—consequently most NIDS installations are configured to alert human operators rather than to intervene directly, even when automated response facilities are available.

The mapping of a typical NIDS to the security component model as illustrated in Figure 3.4 is natural and without significant distortion. Several NIDS implementations use a model of this form as the basis for their architecture: For example, Roesch [112] states that *"there are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem."* A similar architecture is also given by Paxson [98, Figure 1].

The defining characteristic of NIDS, namely the use of raw network data as input, makes these security components reasonably independent of other systems. Consequently NIDS are easily deployed and unobtrusive, usually requiring no modification to existing systems other than finding a suitable point at which network traffic can be intercepted.

The disadvantage of using raw network traffic to monitor applications is that the NIDS analysis has to duplicate the functionality of all intermediate network layers as well as application level abstractions. Such duplication has to be of high fidelity—if not it offers attackers the possibility to *desynchronise* the NIDS from the applications being monitored.

Viewed informally and anthropomorphically, desynchronisation attacks are attempts to sneak past security components: An attacker, aware of the possibility that the target application may be monitored, attempts to avoid triggering the security component by obscuring the attack. An attacker can achieve this by taking advantage of differences between application and security component to formulate the attack in a way that causes it to be interpreted as harmless by the security component but still able to compromise the application.

Desynchronisation attacks have gained prominence in the field of network intrusion detection with the publication of a substantial list of desynchronisation possibilities relating to TCP/IP networks by Ptacek and Newsham [104] and the introduction of desynchronisation tools such as `fragrouter` [127] and `whisker` [107].

A class of systems that can be used to illustrate the problem of desynchronisation are web-enabled applications, including Common Gateway Interface (CGI) programs and their successors.

A substantial number of applications of this type are known to contain vulnerabilities,[8] often input validation flaws. To exploit such a flaw an attacker issues an HTTP [16] request which invokes the application with parameters designed to trigger the execution of attacker specified commands, usually by overflowing a buffer in the case of a compiled language such as C, or by taking advantages of insufficient quoting in interpreted languages such as PERL.

The CGI specification [84] described how parameters may be passed to a web application as part of the URL (parameters are encoded as part of the query string in CGI terminology), an approach which has been adopted by successive systems and formalised in RFC 2396 [17].

This allows an attacker to probe for or compromise a vulnerable application using a single HTTP GET request. Such an approach was used by the Code Red family of worms to propagate, taking advantage of the input validation failure identified in CVE-2001-0500.

If the security components which guard web applications possess information describing known vulnerabilities (for example, that a particular web application fails on receiving a query string exceeding a certain size), then it should be possible to detect attacks by decoding the client requests and matching them against entries in the vulnerability database.

This task is commonly performed by network misuse detection systems. Although this appears simple, it may be vulnerable to desynchronisation at several points:

- Network stack differences or omissions can be used to desynchronise security components. For example, an attacker may interleave additional packets having an invalid checksum or sequence number and carrying a payload which masks the attack. Such packets may erroneously be processed by the NIDS but discarded by the target system. Some attacks of this form can be defended against by implementing a more accurate networking subsystem in the NIDS. However, this may

[8] `Whisker` tests for several hundred weaknesses.

be difficult if parts of a protocol are underspecified—an example of this can be found in the handling of overlapping TCP segments—the specifications do not state which segment of an overlapping pair takes precedence. In such cases a NIDS may have to support all permutations, whereas the guarded system need only implement one. This increases NIDS complexity substantially.

- Environmental differences between security component and guarded system may be exploited by an attacker. For example, a NIDS located several hops in front of the guarded system could be sent a packet having a time-to-live counter low enough to prevent it reaching the guarded system. Such a packet can be used to request the termination of an established session—on receiving it the NIDS may ignore subsequent packets even though they are still accepted by the guarded system.

- Aliases and escapes may allow for the extensive reformatting of attacks. In the case of HTTP requests, examples include `%` escapes which can be used to replace arbitrary characters (*eg* replace the character `;` with `%3b`) and file system aliases (*eg* change `/` to `/./`). Matters may be further complicated by Postel's Principle of Robustness [102] which advises developers to *"be liberal in what you accept, and conservative in what you send."* This advice yields systems that will attempt to correct nonstandard or malformed input in an implementation-dependent fashion—a behaviour that may have to be replicated by the NIDS.

- Flooding and other denial of service attacks may overwhelm the security component. For example, a NIDS located at the gateway of a large, high-bandwidth network may be required to monitor a substantial number of web servers. If the computational resources available to the web server farm exceed those of the NIDS, then an attacker may generate (or wait for the occurrence of) large volumes of traffic to swamp an attack.

- Some systems provide cryptographically secured channels. These may be used by an attacker to evade detection. For example, a web server supporting HTTPS (the cryptographically secured version of HTTP) may provide substantial privacy to an attacker compromising a web application, effectively reducing the function of a conventional NIDS to traffic analysis.

The above desynchronisation possibilities can be described slightly more formally: An application can be approximated by a function f^a which takes

as parameters the current application state s_t^a and some input i_t^a, and computes some output o_t^a and the next state s_{t+1}^a, where t is a discrete measure of time, starting at $t = 0$ and s_0^a is the initial application state:

$$f^a(i_t^a, s_t^a) = (o_t^a, s_{t+1}^a) \tag{3.5}$$

Using the observation of Ilgun [63], one can partition the set of application states S^a into two nonoverlapping subsets, those states in which the application is in a safe state $s \in S_s^a$, and those states which denote a compromise $s \in S_c^a$, where $S_s^a \cap S_c^a = \emptyset$. The characteristic function c^a of subset S_c^a can then be used to test if an application has been compromised.

$$c^a(s^a) = \begin{cases} true & \text{if } s^a \in S_c^a \\ false & \text{if } s^a \notin S_c^a \end{cases} \tag{3.6}$$

A NIDS can be thought of as a function f^n which decides whether an application has entered a compromised state by monitoring application input and output, without having direct access to the internal state of the application.

$$f^n(a, i_t^n, o_t^n, s_t^n) = (d_t^n, s_{t+1}^n) \tag{3.7}$$

Here a is a description of the application, i_t^n and o_t^n are the current observed application input and output, s_t^n the current NIDS state, and d_t^n the NIDS assessment, where $d^n \in \{true, false\}$. Under the conditions that

1. the application description a is sufficiently detailed to derive f^a and the initial application state s_0^a,
2. the inputs observed by the NIDS are the same as received by the application: $\forall t : i_t^n = i_t^a$
3. and the NIDS can encode sufficient state: $|S^n| \geq |S^a|$,

it is possible for the NIDS to construct a shadow copy of an arbitrary application—the state of this copy can then be inspected to determine if the monitored application has been compromised. In the terminology of Bishop [19], a NIDS meeting these requirements can be thought of as a real-time change logging system with access to the initial state of the monitored system.

Desynchronisation becomes a concern when the above constraints are violated. For example, attempts to overload a NIDS can be thought of as violating either constraint 2 (forcing the NIDS to discard some traffic still received by the application) or 3 (exhausting resources allocated to the shadowed

state). Similarly, a NIDS monitoring a cryptographically secured application may fail on constraint 1 (not being party to an initial secret contained in s_0^a) or 2 (lacking access to a copy of the random input used in the generation of cryptographic keys).

It should be noted that for some application classes it is possible for a NIDS to violate the above constraints yet remain synchronised. In particular, an application that contains redundancies can be monitored using a smaller, yet equivalent, shadowed copy by applying compression techniques from the field of information theory. Simplifications are also possible if the the mapping from input and current state to output is amenable to inversion. In such cases the output can be used to detect failures after they have occurred. However, while such simplifications are possible in some cases, they are not necessarily cheap, as both compression and inversion may require a significant domain dependent analysis effort.

For the general case, meeting the above constraints is difficult and unlikely to become easier, given the trends of increased deployment of cryptographically secured channels and the ever greater nesting of protocols. The most substantial example of this is the construction of multiple protocol layers above HTTP, previously considered to be a top-level protocol.

Under such circumstances it remains likely that a typical NIDS analysis module will only receive an approximate description of the higher level interactions between attacker and application. While such approximations may provide analysis modules with sufficient information to detect naive attempts to subvert applications, capable attackers may have enough opportunities to escape detection.

3.3.3 Log and Audit Systems

Subsection 3.3.1 examined infrastructure reference monitors and noted that the infrastructure level security policies enforced by conventional systems are insufficient to protect against the exploitation of flaws in a class of applications termed *trusted*.

This limitation of not being able to absorb the security implications of all application flaws is largely a consequence of keeping the analysis module of a conventional reference monitor simple and making the overall system tamper-proof. Such an approach is justified, as it reduces the possibilities that the reference monitor itself may contain flaws or be subverted. Unfortunately low complexity and tamper-proof often also imply difficult to extend, making it necessary to deploy secondary security components to guard trusted applications.

Network intrusion detection systems, described in the previous section,

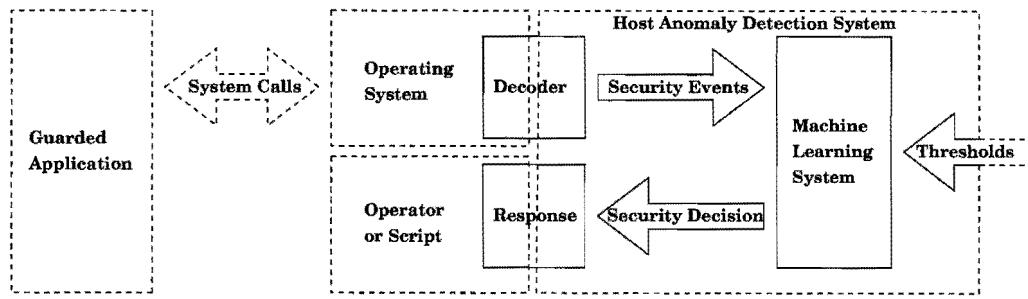


Figure 3.5: Conventional Host Anomaly Detection System mapped to the Security Component Model

are among the more widely used secondary security component types. Although also used to guard against the exploitation of flaws in systems other than applications (*eg* flawed protocol stacks), network intrusion detection systems tend to include substantial analysis functions capable of discovering attempts to exploit flawed applications across a network. The weakness of these security components is that they have to duplicate substantial logic and state to remain synchronised with applications. As this duplication is expensive, most NIDS only provide approximations which makes them vulnerable to desynchronisation and thus of limited effectiveness against non-naive attackers or in cryptographically secured environments.

A strategy used to reduce desynchronisation opportunities involves the closer integration of security components with the systems to be guarded. In terms of the security component model this consists of the partial or complete sharing of data acquisition and decoding functions between guarded system and security component as shown in Figure 3.5. At the implementation level this is usually achieved by instrumenting the systems to report security related information. These security events, in this context known as audit records, are stored for both human and automated analysis. Indeed, most of the early work in the field of intrusion detection [5, 40, 88] involved the analysis of audit data as opposed to network traffic.

A limitation of security components which process audit records is that activity is only reported once it has occurred and that audit interfaces are generally unidirectional. This makes it difficult to take action to prevent attacks, while posthoc responses may lack the facilities to influence the guarded system at the desired level of detail. In other words, responses may be delayed and coarse.

Deciding what information should be logged, and capturing the meaning of this information, presents further challenges. In general it is not feasible to record all system activity. This forces those implementing and manag-

ing an audited system to disregard some information,[9] while the meaning of information which is chosen to be recorded has to be made available to those performing the analysis. When examining this issue, it is useful to differentiate between infrastructure and application level information:

- Infrastructure level instrumentations record security events in terms of the abstractions of the host runtime, logging events such as the system calls made by applications. The Solaris Basic Security Module (BSM) [131] provides a well-known example of such a host audit interface and is used as data source by a number of host based intrusion detection systems including the ones described in [63, 72, 95, 49].

- Application level instrumentations report events relating to the domain in which the application operates. A subsystem often used by applications to report such information is the syslog interface [85], although it is not uncommon for applications to implement their own logging subsystems.

The advantage of using an infrastructure instead of an application audit trail is that the information can be acquired without the co-operation of the application provider and without relying on the integrity of the monitored application. In addition, host audit records have the same structure across different applications, a property which simplifies their manipulation.

The disadvantage of processing infrastructure audit records is that it remains necessary to map this data to higher level abstractions when monitoring trusted applications. Consider the example of a mail client, inherently a trusted application,[10] monitored by a security component which receives a log of the system calls issued by the mail client: Distinguishing reliably between a mailborne worm and a legitimate autoforward function using a system call trace is challenging. Merely examining the sequence of system call names (and ignoring parameters) may not be sufficient, as the propagation of a worm or the autoforwarding of harmless mail may be indistinguishable at that level of detail. Correcting for this by recording more detail, such as the buffer content of IO related system calls, can increase (already nontrivial)

[9] Arguably this problem is more pronounced in host audit systems than in network intrusion detection systems, as exchanges among hosts (and particular those crossing network boundaries) tend to be more expensive and thus represented in a more compact form than the ones occurring within a host.

[10] A mail client satisfies expression 3.2 as it possesses privileges such as the right to manipulate mail folders or the ability to print mail, privileges which should not be available to arbitrary correspondents of its owner, while expressions 3.3 or 3.4 are met whenever the MUA receives or sends a message.

storage and processor demands substantially.[11] Thus few conventional audit subsystems are configured or even designed to do so.

This problem is related to NIDS desynchronisation described in the previous section, in particular the violation of constraint 2—the requirement that the inputs received by guarded application and security component be the same. Security components processing infrastructure audit records may also fail on the remaining constraints, as reconstructing the security status of an application from an audit trail may (in the worst case) impose the same processing demands as running the application. It is worth noting that the use of techniques such as application space threading, scripted extensions, end-to-end encryption and the caching or preloading of resources can contribute to the complication of this task. In other words, systems analysing infrastructure audit data are also vulnerable to desynchronisation, although not to the same extent as network intrusion detection systems: Whereas NIDS share little more with guarded applications than access to the same network, host audit systems provide substantial decoding functions, providing analysts with security events defined in terms of the abstractions that are used to construct or service the guarded applications.

By instrumenting applications it becomes possible to reduce desynchronisation opportunities further: In terms of the security component model, this approach yields access to security events defined in terms of application abstractions. Such a close coupling is difficult to desynchronise, as the security component does not need to duplicate application functionality and has direct access to the application state. However, the use of application level instrumentations also has disadvantages: It requires the co-operation of the application provider and relies on the integrity of the instrumented application. The latter property compels security components operating on application logs to detect attacks before the application is compromised, as subsequent application audit data may no longer be accurate.

A further disadvantage of analysing application-specific abstractions is that it is difficult to construct a substantial analysis framework which spans different application domains. Consequently the systems processing application logs tend to be either specialised systems operating on the logs of a particular application (such as web proxy logs in the case of Calamaris [14]) or are designed to operate on unstructured data (a common approach involves the use of regular expressions to match text log entries, as implemented by systems such as Swatch [57]).

[11] For example, starting the mozilla suite involves approximately 10,000 system calls but generates about 7Mb of IO, even if memory mapped operations are not counted.

3.4 Summary

This chapter has presented a security component model and mapped three of the most widely used security mechanisms to it. The limitations of the mechanisms, when used in the protection of applications, were identified as the following:

- Conventional infrastructure access controls lack the facilities to track the interaction between trusted applications and attackers.
- Low-level intrusion detection systems are vulnerable to desynchronisation because they are required to duplicate substantial application functionality.
- Logging mechanisms tend to be unidirectional and operate after the fact, making it difficult to launch a response. Infrastructure logging systems may also be vulnerable to desynchronisation (albeit in a milder form), while application logs may be unreliable and tend to require substantial domain knowledge during their analysis.

These limitations motivate the investigation into alternative security component designs which may be used to protect applications.

Chapter 4

Taxonomy

4.1 Introduction

The previous chapter considered the strengths and weaknesses of different security mechanisms on an individual basis. This chapter extends the model described earlier into a framework that permits a structured and more general comparison of the tradeoffs made by the designers of various security components. Subsequent chapters will use this framework to classify approaches, identify trends and position a promising hybrid security component design.

The next sections introduce this framework. It takes the form of a taxonomy. The taxonomy is induced by identifying five significant model properties or attributes and using these as dimensions of a classification space. By mapping a given security component to the model, an attribute vector can be derived that places the system at the corresponding co-ordinate in the multidimensional classification space. After its description, the framework will be related to other security taxonomies. An objective of this comparison is to show that the five attributes, intended to describe the tradeoffs which underlie the various designs, are sufficient to cover a significant number of features deemed interesting by other authors.

4.2 Time of Operation

This attribute describes the period over which a given security component is active, relative to the attack it counteracts. Here an attack can be thought of as the interval between first discernible adversarial action and the point at which damage occurs. This divides the time dimensions into three intervals, *viz before*, *during* and *after* the attack.

Each of the phases (collection, analysis and response) of a security com-

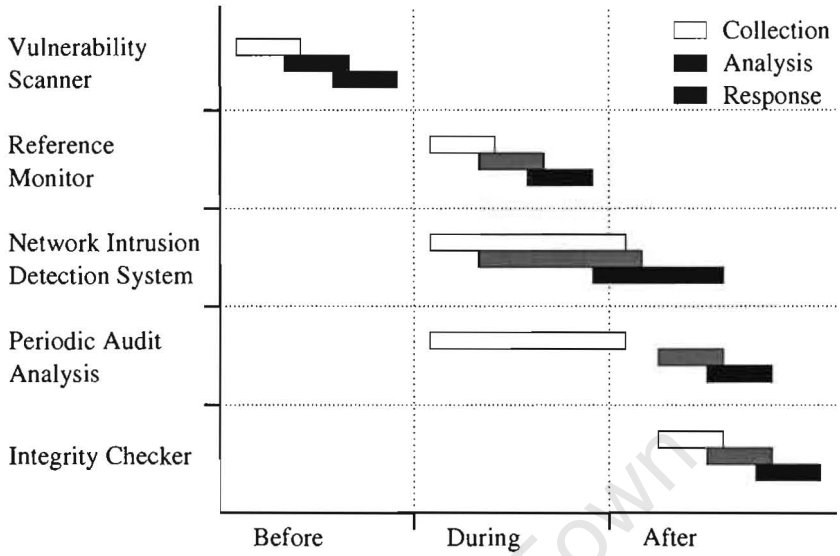


Figure 4.1: Security Components Mapped to the Time Dimension

ponent can be assigned to one or more of these periods, yielding several design permutations. Representative examples are described below and the description is illustrated in Figure 4.1.

- A vulnerability scanner such as NetKuang [144] can be invoked before a system is exposed to attack. Mapped to the security component model all the phases (collection, analysis and response) are active *before* an attack is attempted.

- A reference monitor, and access control systems in general, are deployed during an attack to ensure that the action initiated by an attacker does not progress far enough to damage the guarded system. All the phases of this design are thus active *during* the attack.

- A conventional network intrusion detection system such as `snort` [112] collects and analyses information relating to an attack as it occurs. Although facilities exist to initiate automated responses, the more usual approach consists of alerting a human operator. This design differs from an access control system in that it is not possible to guarantee that any of the phases will complete before the attack succeeds. Consider the example of a buffer overflow exploit sufficiently compact to be contained in a single packet—conceivably such an attack may succeed even before it has been completely decoded by the NIDS. Thus, unlike an access control system, the collection and analysis phases of a NIDS,

although designed to be active *during* an attack, may extend until *after* the attack has completed. The canonical NIDS response (human intervention) tends to occur *after* the attack, although some automated responses may be capable of intervening *during* protracted attacks.

- A periodically invoked intrusion detection system that processes audit records is an example of a security component which collects data *during or after* an attack. This data is stored before being analysed. In other words, analysis and response tends to occur only *after* some damage has been caused.

- An integrity scanner such as `tripwire` [69] collects, analyses and responds *after* an attack has completed.[1] Other members of this category include spyware removal tools and virus disinfectants.

4.3 Location

This attribute describes *where* the security component is positioned relative to the system it protects, whereas the previous attribute described *when* the security component acts relative to the attack.

Since the taxonomy focuses on the protection of applications, it is appropriate to distinguish between security components which are part of applications and those integrated with the infrastructure. In the former case it is possible to differentiate further between applications that are to be protected by the integrated security component and those that host components intended to protect other applications. These three categories can be thought of as placing the security component *inside*, *adjacent to* and *below* the guarded application. The respective positions are denoted by X, Y and Z in Figure 4.2.

The degree to which the security components are coupled to the above systems can be described in terms of the changes that are required to integrate the security component. Here it is possible to distinguish between approaches which make minimal changes to existing systems, those which require changes in configuration, and those which introduce changes at the implementation or design level.

The list below describes the mapping of several common security component designs to the three categories of the location classification dimension. As with the previous attribute, it is possible to map the security model subcomponents to these categories individually.

[1] In this model the acquisition of reference data before the compromise will be considered as being equivalent to a policy statement.

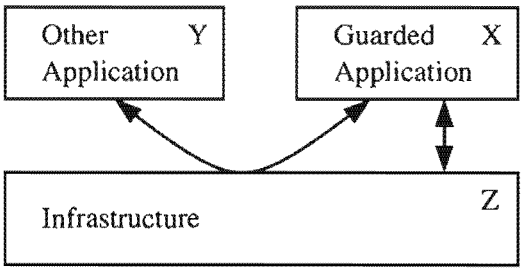


Figure 4.2: Security Component Locations

- A conventional network intrusion detection system processes raw network traffic, placing it at a level below the guarded application. As this data may often be collected without the co-operation or even knowledge of other systems,[2] the collection phase of this security component can be considered to be only weakly integrated with the infrastructure. Likewise the analysis and response stages tend to be largely independent of existing systems.

- A typical host-based intrusion detection system that processes operating system audit records can be thought of as being partially integrated with the infrastructure. Viewed in terms of the security component model, its data collection subcomponent is implemented by the operating system which performs initial data selection and decoding. In contrast, analysis and response tend to be coupled much more weakly, if at all, to existing infrastructure or applications.

- Operating system access controls are examples of security components strongly coupled to the infrastructure, with collection, analysis and response subcomponents usually all deeply integrated with the operating system kernel.

- Application level proxies and gateways [109], of which web accelerators and filters are more recent incarnations, are examples of applications that contain security components intended to protect other applications. For example, a web accelerator may be placed in front of a flawed web server to block attempts to trigger implementation flaws or impose other fine-grained access controls. Usually all phases (collection, analysis and response) of these security component designs are

[2]Usually an effort has to be made to *prevent* the unauthorised collection of this traffic, although capturing the complete traffic stream does present problems in high-speed switched networks and may only be economical with the co-operation of the traffic switching infrastructure.

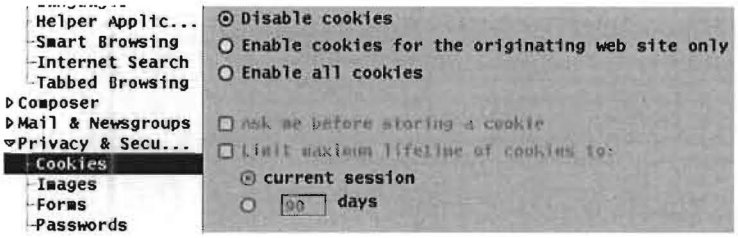


Figure 4.3: A Security Setting of the Mozilla Web Browser

well-integrated with the application gateway, although there do exist designs that only use application gateways to collect information and leave analysis and response to other parties.

- Security components that are designed to protect the applications with which they are integrated occur in a number of forms. Application level logging is an approach which integrates data collection with the application but leaves analysis and response to other systems. Examples of measures that also integrate analysis and response with the application are configuration options which disable application functionality deemed to be dangerous—a screenshot showing a configuration option that disables a web browser function associated with threats to confidentiality, in particular privacy, is given in Figure 4.3. Other examples include consistency checks (a simple example being the use of the `assert` macro to test if security relevant assumptions hold) and application level access controls (such as those implicit in the buddy lists of a chat client which may allow selected individuals greater access to its user).

4.4 Degree of Automation

This attribute describes the extent to which a security component can function autonomously. Informally this attribute describes *who* is needed to provide assistance during the deployment and operation of the system.

Human intervention may however be necessary during all three phases of the model. For each phase the degree of this involvement can be categorised as follows:

- An entirely manual phase relies on a human to perform the given task. For example, an intrusion detection system configured to emit alerts

only (also referred to as a passive system) is a security mechanism that employs a manual response phase.

- Systems providing operational automation perform a given task automatically but require supervision during their deployment or configuration. Usually the management of such systems requires knowledge of the particular environment in which the security component has been deployed.

- Entirely automated security components require little human effort other than that associated with the construction (and activation) of the security component itself.

The following list of security components are examples of designs that require varying degrees of human involvement:

- A logging system (such as the syslog protocol [85]) automates the operational aspects of the data collection phase. The deployment of the collection subcomponent may require some supervision as the insertion of logging calls tends to require domain knowledge. If the system provider has included such instrumentation, this involvement is shifted from the deployment to the construction step. Both analysis and response are left to the likes of system administrators and forensic specialists.

  When a logging system is coupled to a tool such as `swatch` [56] the analysis operation becomes partially automated, although a human is usually needed to manage this phase, primarily to specify the equivalent of a security policy in the form of rules to match events of interest. This involvement can be reduced if a set of predefined rules or signatures has been supplied. Although most analysis tools can invoke arbitrary programs on encountering a matching event, the response facilities are generally used to alert a human—thus the response remains manual.

- A network misuse intrusion detection system such as `snort` is representative of a system which requires little human assistance during the deployment or operation of its collection subcomponent. The operation of the analysis subcomponent is also automated, but human support is required during its deployment, either to construct a set of signatures or select from a set of predefined ones. The objective of analysis subcomponents implementing anomaly detectors (whether host or network-based) is to automate this task entirely. In an ideal case such a system would determine undesirable activity entirely autonomously by deriving the equivalent of a fine-grained security policy from past

behaviour. However, current anomaly detection systems remain dependent on some form of human supervision.[3]

As in the previous example, network intrusion detection systems typically alert human operators instead of mounting automated responses. Exceptions are inlined network intrusion detection systems which are able to discard undesired traffic—in the case of `snort` this functionality is provided by the `hogwash` extension.

- Infrastructure reference monitors in the form of operating system access controls or network firewalls are security components that are designed to operate autonomously—collecting, analysing and responding to attempted transgressions without human intervention. The deployment of such systems does require supervision, as inserting a firewall into an existing network gateway or a reference monitor into an operating system previously lacking access controls can involve extensive design changes. However, as access control mechanisms are usually integrated with modern network elements and operating systems, this task does not have to be undertaken on a per-installation basis. The effort that remains dependent on a particular environment is the management of the analysis module, ie the specification of an access control policy.[4]

- A system such as `stackguard` [37] is an example of a security component which can function with almost no supervision. Except for the initial deployment effort of recompiling the guarded executable (a task which requires little analysis effort and can be performed by an application provider), little human involvement is required—once instrumented, this security mechanism will monitor a process for failures that result in stack corruptions and will abort execution if any are detected.

4.5 Analysis Abstraction

This section provides an initial categorisation of the events submitted for analysis. Put simply, this section describes *what* the analysis subcomponent operates on.

[3]Human involvement is often needed to supply annotated training data, tune learning parameters and, most significantly, disambiguate between true and false positives [9].

[4]In this context a role-based access control approach provides a level of indirection which isolates core security policy statements from some environmental dependencies and thus reduces human involvement.

As in the previous sections, the intention is not to establish a detailed framework capable of defining every possible abstraction uniquely—arguably such an effort requires further advances in the representation of ontological information. Instead the objective is to identify a general quality of the entities in terms of which the security events are defined. Three classification categories are proposed for this purpose:

- *Infrastructure* abstractions describe events in terms of entities which are *used* to construct or service the application. Examples include host operating system *processes identifiers*, *files* and *system calls.*

- *Application* abstractions are introduced by the designer of a particular application to describe the domain in which the system operates. For example, a word processor might generate security events defined in terms of *paragraphs* and *chapters.*

- *Security* abstractions are independent of a particular application or infrastructure but are introduced to capture a generic security property. For example, a *security class* of a multi-level access control system labelled *confidential* is a construct which could be used to describe both a word processor paragraph or an operating system file.

The abstraction dimension differs from the placement or location attribute described earlier—for example, designs exist which place a security component inside an application, yet report security events in terms of infrastructure abstractions; a simple case is a modification that causes an application to log the system calls it issues.

Of interest is that the application and infrastructure categories are easily accessible to those providing the respective systems, as the same conceptual frameworks used to construct the systems can also be used to perform the security analysis. Consider the example of a network router which operates on an abstraction involving packets and networks. Constructing an analysis subcomponent that processes security events in the form of packets sent to networks does not require the introduction of an alternative conceptual framework. Similarly, an application-specific abstraction permits the application programmer to generate security events without a translation effort.

In contrast, the security abstraction category requires a mapping of the reported security events from the native abstraction to an alternative form prior to analysis. A number of these mappings are described below:

- The conventional access control model (subjects having a set of access rights to objects) can be considered a security abstraction which maps

security-related activity of a particular system to a set of *subject-object-access triples*. Usually the meaning of these values remains dependent on a particular domain (an operating system may operate on files, a database on tables, *etc*); however, when a security labelling scheme (either in the form of security compartments or levels) is used, an access control abstraction can be separated from the application domain.

- Analysis components employing machine learning techniques to discover anomalies (an approach sufficiently general to be applied to both applications and infrastructure activity) tend to require particularly *information-dense representations* of security relevant information for optimal operation. Mapping system activity to such a representation usually involves feature selection (see [79] for an examination of this task in a security context) and encoding efforts (an example of a mapping from a TCP connection attempt to a compact 49 bit string is described in [61]).

- The syslog interface is primarily used to report unstructured infrastructure or application-specific events. However, it also includes an eight-valued *severity* field (ranging from routine debug and informational messages to high-priority alerts and emergencies) which makes it possible to filter events without knowledge of the domain in which the reporting system operates.

- The Distributed Auditing Standard (XDAS) [133] defines 9 default event classes, which in total contain 45 generic events to which the activity of a given system can be mapped. Event classes relate to tasks such as the administration and use of accounts, communications channels, services and applications. Within each of the event classes, activity is generally reported in terms of the *creation*, *access*, *modification* or *destruction* of an entity.

- Although the Message Exchange Format of the Intrusion Detection Working Group (IDWG) [43] as well as the Logging Data Map (LDM) of Ranum and Robertson [108] operate on domain abstractions, primarily network related, it is possible to identify a security abstraction (not unrelated to the subject-object-access model) which defines activity in terms of a *source* (possible attacker) and a *target* (potential victim). In the case of the LDM, 10 of the 23 tags are used to describe source and target entities (example tags are `SRCPID`, `SRCPATH`, `SRCDEV` and `SRCUSER`), while others are used to label information such as event priority, time and error condition. Similarly alerts defined using the

proposed IETF Intrusion Detection Message Exchange Format consist primarily of source and target elements, which may be nodes, users, processes or services.

4.6 Analysis Complexity

This attribute categorises the computational requirements of the analysis phase. The thesis of Kumar [75] describes the complexity of encoding misuse signatures in terms of four major categories, in ascending order of complexity:

$$ExistenceTests \subset SequenceMatches \subset ExtendedRegularExpressions \subset OtherPatterns$$

An alternative approach would be to describe the time and space requirements of a security component directly. However, given that the above categories are established, representative of mechanisms used operationally, and can be mapped to the time and space requirements, the four categories are also used in this taxonomy. Members of these categories are given below:

- The reference monitors of commonly used operating systems are designs that belong to the first category: In order to test if an access request should be granted, a lookup is performed to establish if the requested access right is available for the given subject/object pair—in other words, the reference monitor tests if a given privilege exists. Both the time and space requirements of this approach are low: Usually no state needs to be maintained across events and the test can often be performed in constant time.[5]

- The approach of matching short sequences of system calls, as described in [61], is representative of the second category, provided that the generation of sequences during the training phase is regarded as a variant of policy formulation, rather than part of the immediate analysis.

- Regular expressions have been used for a number of analysis tasks—although usually associated with the processing of unstructured security event streams, regular expressions have also been applied in the specification of behaviour restrictions on applications [121] or generated automatically as anomaly detectors [92].

[5]Consider the request by a unix process to open a file: Comparing the process owner against the file owner and permission bits requires only a few additional memory references.

- Analysis components belonging to the fourth category include those that provide extension interfaces or policy interpreters for computationally complete languages. DEEDS [3] is an example of the latter, as policy is stated in the form of a Java program. Such an analysis can have unbounded time and storage requirements.

The above categories were not intended to describe the complexity of anomaly detection systems. If anomaly detectors are thought of as systems that automate the specification of policy, then this function (and its complexity) is distinct from the matching of the learnt patterns against the security event stream. Otherwise the control logic and state size of anomaly detection systems is usually sufficiently large to assign them to the latter categories. Similarly human analysis is regarded as belonging to the most complex category.

4.7 Related Taxonomies

This section relates the security component taxonomy introduced in this chapter (and abbreviated to SCT in this section) to other classification work in the area of computer security. The largest effort in this area has been directed at categorising security flaws and the attempts to exploit them—taxonomies of this type have been described in [78, 81, 74, 62, 48], amongst others. Despite being criticised by a number of authors for being somewhat ambiguous [62] or nonspecific [74], the taxonomy of Landwehr [78] remains an appealing framework. It proposes three security flaw dimensions:

- The *genesis* attribute describes how flaws are introduced, defining two major categories that differentiate between flaws that have been introduced inadvertently and those inserted deliberately.

- The *time of introduction* positions the point at which the flaw is inserted relative to the lifecycle of the system—this attribute defines the major categories of development, maintenance and operation.

- The *location* describes which systems contain the flaw. Here the major categories are hardware and software, with the latter being further divided into operating system, support and application subcategories.

This flaw taxonomy can be used to position the work of this thesis—as motivated in a previous chapter, the focus is on applications which contain flaws introduced accidentally. In terms of the Landwehr taxonomy, SCT is intended to describe the following security flaw set:

- *Genesis*: Inadvertent or nonmalicious
- *Time of introduction*: All categories, from development to operation
- *Location*: Applications

Although useful in positioning and evaluating security components, flaw and attack taxonomies provide only limited information about the mechanisms used to defend computer systems—such information is better conveyed by direct descriptions of security components. The following subsections examine such taxonomies and relate them to the framework proposed in this chapter.

The comparison is qualified by the observation that the various taxonomies do not have identical subject domains, for example some taxonomies focus more on collection mechanisms, while others seek to classify responses.

4.7.1 Conventional IDS Classification

A simple taxonomy, which is frequently used to introduce intrusion detection systems, consists of two classification attributes—one dimension distinguishes between *anomaly* and *misuse* detectors, the other between *network* and *host-based systems*.

Mapped to SCT, the distinction between misuse and anomaly detection systems can be described using the *degree of automation* attribute (see Section 4.4) applied to the *analysis* phase: A misuse detection system can be considered an example of a security component which contains a partially automated analysis subcomponent: A security administrator is no longer required to analyse security events directly, instead a security expert compiles a set of signatures that describe undesired activity and drive the analysis. Anomaly detection systems are examples of security components that aim to automate the analysis phase further: Instead of specifying policy manually, a corpus of past activity is submitted to a machine learning system which uses this data to derive the equivalent of a security policy.

The distinction between network and host-based systems can be thought of as an alternative categorisation of the *location* attribute (see Section 4.3) of the security component model. Whereas the conventional approach groups applications and operating systems (as well as related support systems such as virtual machines and language interpreters) into the single *host* category, SCT groups networks and operating systems into the *infrastructure* category. The mapping between these two partitions is illustrated in Figure 4.4.

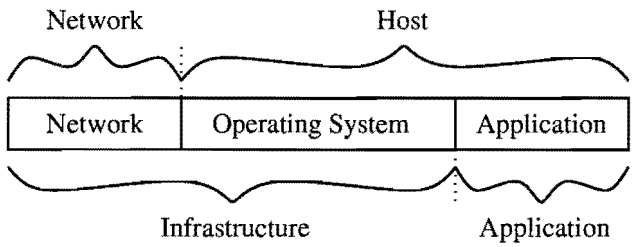


Figure 4.4: Conventional and SCT Partitions of the Location Dimension

4.7.2 Zamboni Data Collection Classification

Zamboni [142, chapter 2] examines a number of intrusion detection architectures and classifies these, primarily in terms of their data collection phase. Two classification dimensions (termed *conceptual classifications* by their author) are identified:

- The *collection structure* is used to distinguish between *distributed* and *centralised* data acquisition. A collection structure is defined as distributed if the number of collection locations is *"directly proportional to the number of monitored components"*. The same division is also used to describe the subsequent analysis.

- The *collection mechanism* distinguishes between *indirect* and *direct* monitoring, with the former category being defined as *"the observation of the monitored component through a separate mechanism or tool"* and the latter as *"the observation of the monitored component by obtaining data directly from it"*. Observation mechanisms of the direct category are further divided into *external* and *internal* sensors. An external sensor is defined as *"a piece of software that observes a component in a host and reports data usable by an intrusion detection system, and that is implemented by code separate from that component"*[142, pg 16], whereas an internal sensor is *"a piece of software that observes a component in a host and reports data usable by an intrusion detection system, and that is implemented by code incorporated into that component"*[142, pg 17].

The *collection mechanism* dimension of this taxonomy can be mapped to the SCT *location* dimension (described in Section 4.3) as follows:

- A sensor *internal* to an application can be regarded as a security component which locates its collection, and possibly analysis and response, subcomponent *inside* the guarded application (point X in Figure 4.2).

- The distinction between an IDS using an *indirect* collection mechanism and one using an *external* sensor as defined above seems slight: Monitoring a component using a *"separate mechanism or tool"* appears to be little different from receiving data from something which is *"implemented by code separate from that component"*. For this reason both these categories will be considered to be equivalent to security component designs whose collection subcomponent is placed *adjacent* to or *below* the monitored component/guarded application (points Y or Z in Figure 4.2).

The SCT *location* dimension can also be used, in part, to describe the *collection structure*, by noting that any design which locates the collection subcomponent within the guarded application has a distributed collection structure as the number of collection subcomponents is, of necessity, proportional to the number of monitored components/guarded applications.

4.7.3 AINT: The Anti-Intrusion Taxonomy

AINT, introduced by Halme [55], informally describes a number of mechanisms used to defend computer systems. The taxonomy does not identify distinct classification attributes or dimensions, instead it lists the following six categories:

1. Prevention *"... seek[s] to preclude or at least severely handicap the likelihood of success of a particular intrusion."*

2. Preemption *"... techniques strike offensively prior to an intrusion attempt ..."*

3. Deterrence *"seeks to make any likely reward from an intrusion attempt more troublesome than it is worth."*

4. Deflection *"dupes an intruder into believing that he has succeeded in accessing system resources, whereas instead he has been attracted or shunted to a specially prepared environment for observation".*

5. Detection *"encompasses those techniques that seek to discriminate intrusion attempts from normal system usage and alert the SSO".*

6. Counteraction *"empower[s] a system with the ability to take autonomous action to react to a perceived intrusion attempt."*

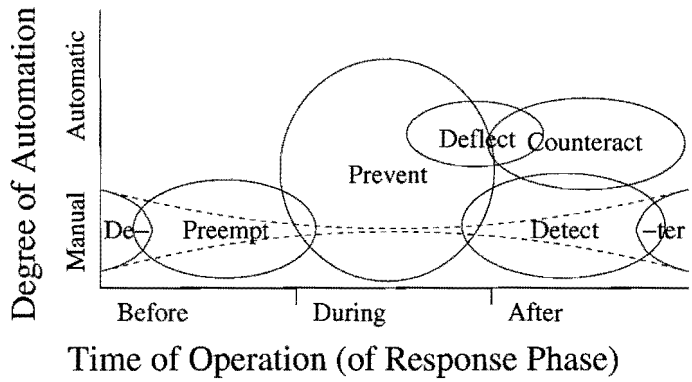


Figure 4.5: AINT Categories mapped to the SCT Time and Automation Attribute of the Response Subcomponent

Halme notes that the deflection category (of which honeypots and lightning rod accounts are given as members) can often be regarded as a special type of intrusion countermeasure.

Given this overlap, it is possible to describe the AINT model categories in terms of SCT *time of operation* and (to a lesser extent) *degree of automation* attributes as applied to the *response* phase in the following way:

Preemptive measures (Halme lists the infiltration of attacker organisations as an example) tend to involve manual intervention *before* an attack is attempted. Preventive measures aim to intercede before damage has been caused and are thus active *before* and *during* an attack. Deflection can be seen as a specialised countermeasure which either prevents or mitigates attacker damage, while the more general detection and countermeasures are generally active *during* and *after* an attack. The distinction between countermeasure and detection is that the response of the former is at least partially automated, while the latter response is manual. Lastly, a deterring response is initiated not to block or undo the damage of the current attack but to head off further ones. An approximate illustration of these mappings is given in Figure 4.5.

4.7.4 Fisch and Carver Response Taxonomies

A paper by Carver [28] describes two taxonomies which aim to classify possible intrusion responses. The first, developed by Fisch at the same institution as Carver, is described as consisting of two primary features:

- The *time* when a compromise is detected. This attribute is divided into two categories: *During* and *after* an attack.

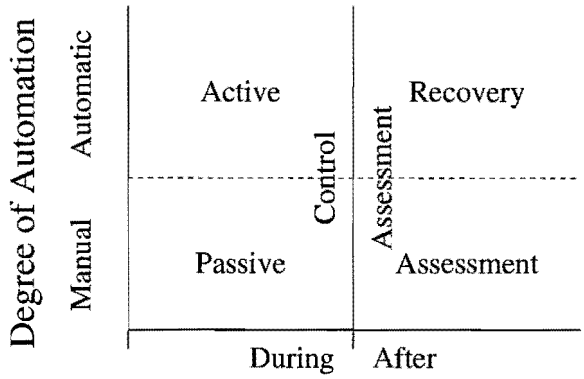


Figure 4.6: Features identified by Fisch mapped to the SCT Time and Automation Attribute of the Response Subcomponent

- The *goal* of the response. Relative to the time attribute two categories are identified: *Control* mechanisms are active during an attack, while *assessments* are undertaken afterwards. The former category is subdivided into *active* and *passive* control mechanisms, while the latter distinguishes between *assessment* and *recovery*.

The main characteristics of the Fisch Damage Control and Assessment Taxonomy can be mapped to two attributes of the Security Component Taxonomy proposed in this document, namely *time* and *degree of automation* particularly if related to the *response* phase. This mapping is shown in Figure 4.6. Here passive or assessing systems perform data acquisition and analysis tasks, but do not provide an automated response component.

The second taxonomy (Carver's Intrusion Response Taxonomy), identifies six classification dimensions and associated categories:

1. Timing of the Attack: Preemptive, During and After
2. Type of Attack: Availability, Confidentiality and Integrity
3. Type of Attacker: Cyber Gangs, Economic Rivals, Military Organisations, ...
4. Degree of Suspicion: Low to High
5. Attack Implications: Low to Critical
6. Environmental Constraints: No offensive response, no router resets, ...

With the exception of the time dimension, which has an equivalent in the first attribute of the taxonomy proposed in this chapter, the dimensions tend to enumerate factors which modulate responses, rather than describe the response mechanisms directly and are thus not examined further in this thesis.

4.7.5 Axelsson's IDS Taxonomy

In introducing a survey of intrusion detection systems, Axelsson [10] presents an IDS taxonomy. In addition to the distinction between anomaly and misuse detection systems, the following eight attributes are identified:

1. Time of detection: This dimension distinguishes between systems that are real-time and those that are not.
2. Granularity of data-processing: This attribute describes whether a system operates on batches of data or a continuous data stream.
3. Source of audit data: As in the case of the conventional IDS taxonomy, the primary distinction is between network and host-based data collection.
4. Response to detected intrusions: This dimension distinguishes between systems that are active (respond autonomously) and those that are passive (notify an authority).
5. Locus of data-processing: This divides systems into designs that process data centrally or distribute the task over multiple locations.
6. Locus of data-collection: Like the previous dimension, this attribute distinguishes between centralised and distributed designs.
7. Security: The ability of the IDS itself to withstand attack.
8. Degree of Interoperability: The extent to which the system under consideration can interoperate with other intrusion detection systems.

The first two classification attributes relate to the time over which an IDS is active. Given a definition of real-time which requires that an attack be detected within a certain time period, and a batch size metric describing the interval over which events are collected, these two classification attributes can be related to the SCT *time of operation* dimension as follows:

The first dimension (time of detection) can be mapped to the interval between the occurrence of an attack and the completion of the analysis phase.

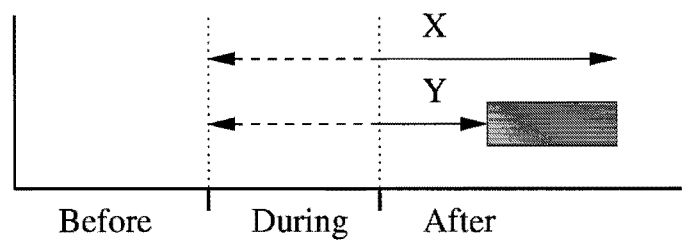


Figure 4.7: Time of Operation of Analysis Phase

The second dimension (granularity) provides an indication of the delay between the time at which an attack occurs and the point at which analysis commences. These intervals are illustrated in Figure 4.7 by X and Y respectively, with the dashed line segments indicating the period during which events of interest occur.

This interpretation may also help to clarify the assertion by Axelsson which states that the first two attributes *"do not overlap since a system could process data continuously with (perhaps) considerable delay, or process data in (small) batches in 'real-time'."* The two attributes do indeed measure different timing aspects but are not completely unrelated, as the batch length is bounded by the interval during which analysis has to be completed. Put simply, a system required to discover a given event within N seconds of its occurrence should be supplied with data at intervals smaller than N seconds.

The third classification dimension of the Axelsson taxonomy describes the audit data source. It corresponds to the second attribute of the conventional IDS taxonomy described in Subsection 4.7.1 which, in turn, can be mapped to the SCT *location* attribute as illustrated in Figure 4.4.

The response attribute, dimension four, can be related to the SCT degree of automation attribute (described in Section 4.4) of the response component, with a passive response being equivalent to a manual response phase, and an active one being partially or completely automated.

Dimensions five and six are used to distinguish between distributed and centralised designs and have been incorporated into the Zamboni taxonomy described in Subsection 4.7.2. Although the taxonomy proposed in this chapter does not consider the issue of distribution directly, the SCT location dimension identifies designs (collection, analysis or response mechanisms resident in the guarded application) which are of necessity distributed. In addition, the SCT analysis abstraction can be used to identify classes of systems (those operating on domain independent abstractions) that are amenable to centralised analysis even if collection and response subcomponents are heterogenous—an issue also closely related to attribute eight of the

taxonomy summarised in this section, the degree of interoperability.

Dimensions four, five and six can be used to illustrate how the approaches taken to derive the Axelsson taxonomy and the one introduced in this chapter differ: Of interest is that the former taxonomy distinguishes between distributed and centralised collection and analysis (attributes six and five) but does not include a similar attribute for the response phase. Conversely Axelsson differentiates between systems that provide an automated response and those that defer to a human (attribute four) but does not do the same for the collection and analysis phases.

These asymmetries of the Axelsson taxonomy are induced by contemporary intrusion detection designs. The systems surveyed by Axelsson automate the operational aspects of the collection and analysis phases but only a few respond autonomously. This makes it it is possible to ignore distributed response or manual collection categories when classifying conventional intrusion detection systems. However, for the purpose of exploring the set of possible security component designs, a taxonomy derived from a set of existing intrusion detection systems may be limiting, hence the approach of deriving the security component taxonomy from an abstract model.

4.8 Summary

This chapter has used the security component model to derive a taxonomy of systems which protect applications. The taxonomy uses the attributes *time*, *location*, *automation*, *abstraction* and *complexity* to generate a five-dimensional classification space. In the next chapter, this classification space will be used as framework to describe the tradeoffs which underlie various security component designs.

In order to examine the coverage of the framework, it was related to a number of existing taxonomies. These included an informal categorisation (Halme), several narrower approaches (Zamboni focused on the data collection phase, while both Fisch and Carver considered possible responses) and a more extensive one (Axelsson). In each of the cases it was shown how most of the features identified by their respective authors could be described in terms of the five proposed attributes.

Chapter 5

Comparison

5.1 Introduction

The previous chapter introduced a taxonomy of security components in the form of a five-dimensional classification space. This chapter assigns different security component designs to regions in this space. In conjunction with the category examples provided earlier, this can be seen as an augmented review of related work—whereas a conventional review would simply enumerate systems, this chapter attempts to arrange and group the various approaches in order to explore the tradeoffs which underlie the designs and identify alternative approaches.

The comparison is divided into two parts: A section which examines infrastructure security mechanisms and a section which considers measures operating at the application level. Viewed in terms of the taxonomy, the two sections can be thought to divide the classification space using a plane orthogonal to the location dimension (refer back to Section 4.3 for a description of this dimension).

5.2 Infrastructure Security Components

This section examines those security components which are part of the general purpose infrastructure below the level of the application. Most classical security mechanisms belong to this group.

5.2.1 Infrastructure Reference Monitors

Arguably the oldest and most established security component design is the infrastructure reference monitor. Most operating system access controls are

representatives of this design. However, as the taxonomy regards both networking and operating systems as being part of the infrastructure, simpler network firewalls also belong to this category.

The subspace occupied by these security components is given below:

Time of operation:	During the attack
Location:	Infrastructure
Degree of automation:	Automatic operation, manual policy specification
Analysis abstraction:	Infrastructure
Analysis complexity:	Low (existence tests)

Anderson's seminal description [4, page 22] of this security component design outlines the requirements of a security kernel implementing a reference monitor as follows:

1. *"The reference validation mechanism must be tamper proof"*
2. *"The reference validation mechanism must always be invoked"*
3. *"The reference validation mechanism must be small enough to be tested (exhaustively if necessary)"*

These requirements influence the position of reference monitors in the classification space. A security kernel implies an infrastructure component, the minimality requirement suggests an analysis subsystem which is of low complexity, and the tamper-proof requirement calls for a system resistant to extension and thus limited to infrastructure abstractions.

Essentially this design trades expressive power for robustness. As noted previously, this tradeoff allows infrastructure reference monitors to provide substantial and reliable protection to *trusting* applications but makes them ill-suited to follow higher level exchanges between *trusted* applications and potential attackers.

It can be argued that this limitation, as well as the cost of manually specifying a sound access policy, has motivated the introduction of additional security component designs such as audit and intrusion detection systems.

5.2.2 Manual Infrastructure Audits

Given the limitations of reference monitors, a class of components has been developed which allows security specialists to analyse computer systems for transgressions not blocked by infrastructure access controls. Systems of this

type include audit systems that collect data from the operating system for review (the Solaris Basic Security Module (BSM) [131] is a well-known representative) as well as firewall logs or a system such as the Packet Vault [7] which perform a similar function at the network level.

Because humans operate at lower speeds and have shorter concentration spans than computers, such analysis is generally performed after the fact and intermittently. This security component design can be mapped to the taxonomy as follows:

Time of operation:	Collection during and after the attack, analysis and response *after* the attack
Location:	Collection at infrastructure
Degree of automation:	Automated collection, manual analysis and response
Analysis abstraction:	Infrastructure
Analysis complexity:	High (performed by humans)

These systems can be thought of as security components which perform a much more sophisticated analysis than conventional reference monitors but, because of the operational involvement of human security analysts, trade this for a greater delay and higher cost.

5.2.3 Conventional Intrusion Detection Systems

Conventional intrusion detection systems examine either operating system audit data or network traffic in place of expensive and slow human analysts.

If a machine learning system is employed, both the operational aspects as well as the policy specification are automated, otherwise policy in the form of a set of misuse signatures is specified by hand.

Oriented in the security component classification space, a conventional intrusion detection system can be regarded as an intermediate between an operating system reference monitor and a manual audit analysis with respect to the time of operation and analysis complexity dimensions: An IDS is faster than a human, but slower than an access control system because it is unable to delay an attacker while the analysis is conducted. Similarly, an IDS analysis subsystem tends to be more complex than the simple lookup mechanism of an access control system but is not as sophisticated as a human.

Time of operation:	Collection during the attack, analysis and response during or after the attack
Location:	Collection at infrastructure
Degree of automation:	Automated collection and analysis, automated policy specification in the case of anomaly detection, manual response
Analysis abstraction:	Infrastructure
Analysis complexity:	Intermediate to high (usually matching sequences or better)

5.2.4 Extended Infrastructure Reference Monitors

In the last decade a number of intrusion detection projects have developed systems which can be considered augmented reference monitors. At the operating system level, such projects include Janus [51], the Wrapper Support System (WSS) [47], the Auditing Specification Language (ASL) [121], pH [126] and Medusa DS9 [143], with recent efforts such as the Linux Security Module (LSM) Interface [140] aiming to standardise and popularise these extensions.

The network equivalents of these systems are more sophisticated firewalls. These are often advertised as *stateful* or *inspecting*, of which the ftp protocol parser of the Linux `netfilter` is a well-known example. Also belonging to this class are *inlined* network intrusion detection systems which are able to discard undesirable traffic, an example being the `hogwash` extension to `snort`.

In terms of the security component taxonomy, these systems can be seen to increase the complexity (and thus the analytical power) of conventional reference monitors without delaying the response until after an attack has succeeded, as is the case in conventional intrusion detection systems.

Time of operation:	During the attack
Location:	Infrastructure
Degree of automation:	Automated collection and analysis, automated policy specification in the case of anomaly detection
Analysis abstraction:	Infrastructure (possibly applications)
Analysis complexity:	Intermediate to high (usually matching sequences or better)

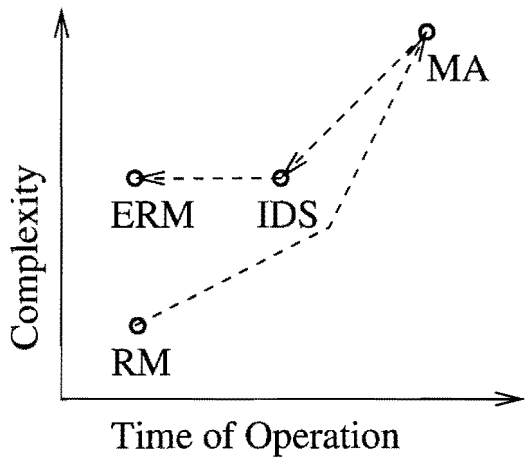


Figure 5.1: Complexity versus Timeliness: Comparison for Infrastructure Reference Monitors (RM), Manual Audits (MA), Intrusion Detection Systems (IDS) and Extended Reference Monitors (ERM)

5.2.5 Analysis

Although somewhat of a simplification, the four infrastructure security component designs summarised in the above classification can be thought to form a trajectory (illustrated in Figure 5.1). Here human analysts compensate for limitations of classical access control systems, intrusion detection systems replace the manual effort, and extended reference monitors intervene sooner when prevention is possible.

This progression is particularly pronounced in the case of operating system security components, where the initial IDS model of Denning [40] proposes using the reference monitor subject-object-access triple[1] as analysis data. This can be viewed as the interface between access control and intrusion detection. A similar transition can be noted in IDS designs. Earlier implementations (see [88] for a survey) mimic human analysis by being invoked periodically, later systems operate on a near-realtime basis, and recent systems act as preventive reference monitor extensions.

With regard to the equivalent network security components, this progression is not as pronounced. Nevertheless, a trend towards the extended reference monitor design remains apparent[2], involving either a direct progression from simple to complex firewall or an indirect one involving the use of inlined network intrusion detection systems to perform what amounts to

[1] Augmented with a timestamp, resource usage and an error status.

[2] Note that this observation is restricted to infrastructure components. Application proxies are examined in the next section.

an access control function.

Incorporating intrusion detection functions into reference monitors allows for a more effective response, because blocking damaging activity is usually more easily accomplished and automated than damage recovery. This benefit is traded against:

- An increased sensitivity to false positives, because mistakenly blocking access may be costly.
- Stricter time constraints, given that a reference monitor usually lies on the critical path of a system and thus has an impact on overall system latency.[3]

The increased deployment of more sophisticated firewalls and other extended reference monitors suggests that components which act before an attack succeeds are attractive. This implies that the above disadvantages are acceptable in the protection of real-world systems. However, the fact that many of these extended reference monitors (particularly those requiring changes to operating systems) have evolved from intrusion detection efforts rather than being direct reference monitor advances, can be interpreted as a reluctance to violate the simplicity requirement of classical reference monitor designs. Similarly the tamper-proof requirement can be seen as having delayed the widespread introduction of such systems until the adoption of open-source and modular kernels made arbitrary third-party operating system extensions feasible.

As motivated previously, a significant reason for the introduction of more complex infrastructure security components is the need to regulate higher level exchanges. In such cases a substantial part of the infrastructure security component may be dedicated to the re-implementation of application functions. Examples include duplicated application protocol parsers and file format handlers, as built into inspecting firewalls (for example to track incoming ftp data connections and open the appropriate port) or mail transfer agents (to scan documents, which are normally considered opaque at that level, for viruses and worms). This thesis considers the trend toward such re-implementation suboptimal, since it is:

- Inelegant: It blurs the layered design of computer systems.

[3]Throughput remains unaffected when compared to an intrusion detection system of equivalent complexity, if an overloaded IDS that discards data is considered equivalent to a reference monitor that is only invoked occasionally.

- Dangerous: A failure of an infrastructure security component *may* have greater security implications than one affecting the application it guards, particularly as a number of protection mechanisms may not be available at lower levels.[4]

- Demanding: The duplication has to be performed with a high degree of fidelity. It may require the tracking of substantial state, differences among versions or deviations from documented behaviour. Failure to do so may allow attackers to evade detection or (in the case of automated responses, as encountered in shunning firewalls or traffic normalisation systems) stage denial of service attacks.

- Not always possible: End-to-end or application level cryptographic measures are explicitly designed to prevent the acquisition of information by lower level components.

These limitations motivate the investigation into application security components undertaken in the next sections.

5.3 Application Security Components

The alternative to increasing infrastructure security component complexity is to shift security components into applications where domain knowledge does not have to be duplicated. In terms of the security component taxonomy, these two different approaches develop the simple reference monitor design along different dimensions: The former requires a more complex analysis, while the latter depends on the integrity and support of a larger system by being located within applications. The two different strategies can oriented in the classification space as shown in Figure 5.2.

This section reviews and categories application level security component types and identifies several possible reasons which have made them a less popular choice than infrastructure systems.

[4]Consider the example of the ftp protocol parser part of a network firewall: A failure in the parser, often a kernel resident system (where application resource limits, address space restrictions and a lowered user id do not apply), may allow an attacker complete access to and control over the traffic reaching a network. In contrast, a compromise of the ftp server may have only yielded access to a sandbox containing public data, given that a number of anonymous ftp server implementations for unix systems `chroot` into a directory subtree and change to an unprivileged user id.

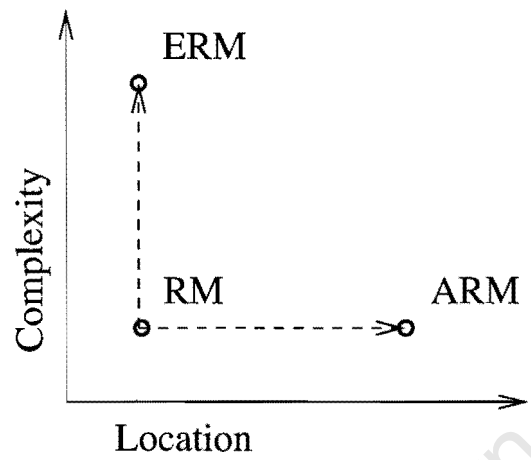


Figure 5.2: Expansion of Conventional Infrastructure Reference Monitors (RM) toward Greater Complexity (ERM) or the Application Level (ARM).

5.3.1 Vendor-Supplied Security Components

Security components of this type are constructed as part of the application development effort. Examples include internal consistency checks, configuration options which disable high-risk operations, as well as application resident access control systems (refer back to Figure 4.3 for an example). Application providers usually possess a good understanding of the domain abstractions but are likely to allocate only limited resources to the security effort, for reasons examined in Chapter 2.

Such constraints yield security components which are usually less well developed than their infrastructure counterparts. Most affected is the analysis phase which tends to be of low complexity and static, with policy changes often requiring an application restart or even a recompilation.

These characteristics can be mapped to the classification space as follows:

Time of operation:	Variable, often before or during the attack
Location:	Application
Degree of automation:	Manual insertion and management
Analysis abstraction:	Application specific
Analysis complexity:	Variable, often simple

5.3.2 Application Logging

Although application logging subsystems can be viewed as a special case of vendor supplied security components, they are examined separately. This

is because these systems transfer the analysis and response effort from the application provider to an administrator or security specialist. Application logging systems differ further from other vendor security components (such as internal consistency checks) in that they tend to be active after the fact, reporting events that have occurred recently. This is a limitation, since a successful attack may disable logging or even generate false entries.

Time of operation:	Collection during the attack, analysis and response only after the attack (unreliable)
Location:	Collection in the application
Degree of automation:	Manual insertion, analysis and response, automated collection
Analysis abstraction:	Application specific
Analysis complexity:	Variable

5.3.3 Application Proxies

Application proxies, both classical (*eg* the TIS's firewall toolkit [109] or `fk` [71]) and alternative (*eg* privacy enhancing proxies such as `junkbuster`), operate adjacent to the guarded application. However, while application proxies are widely used to cache data, pure security proxies have in a number of cases been displaced by infrastructure systems (*eg* network level firewalls which tracking ftp transactions) or application resident security components (*eg* the `mozilla` and `galeon` web browsers acquiring privacy enhancing features). Performance concerns may account for this or (in the latter case) a desire to reduce the amount of duplicated effort.

Time of operation:	During the attack
Location:	Adjacent to the application
Degree of automation:	Manual construction, otherwise automated
Analysis abstraction:	Application specific
Analysis complexity:	Variable

5.3.4 Automated Security Tools and Components

For reasons provided earlier it is not always possible to rely on application provides to include sophisticated security components in their products, while both the construction and operation of application proxies may be inefficient.

These issues have motivated the introduction of utilities which insert security components into applications automatically. Well-known representatives of this approach are `stackguard` [37] and `RAD` [33] that block a particularly common type of input validation failure. Although not operational at runtime, static analysis tools such as Flawfinder [139] and RATS [123] also are intended to prevent attacks by flagging risky constructs (such as the use of `strcpy` in place of `strncpy`).

The limitation of completely automated security components is that they are intended to be used on a wide range of applications. They thus need to operate on common abstractions, typically the entities used in the construction of the applications (functions, objects) or generic runtime measurements (execution time, memory use). As these are generic, it is difficult to capture the domain knowledge and other features specific to a particular application.

Thus while these components may be located inside (or operate on) applications, it is possible to view the the analysis abstractions as being located close to or even within the infrastructure. These characteristics place automatically added application security components in the following region of the classification space:

Time of operation:	Before (static analysis) or during (runtime component) the attack
Location:	Application
Degree of automation:	Automatic insertion and operation
Analysis abstraction:	Near infrastructure
Analysis complexity:	Variable, possibly complex

5.3.5 Expanded Security Infrastructure

The fact that the fully automated security components described in the previous sections operate on abstractions common across applications makes it possible to absorb these measures into the infrastructure. In other words, the infrastructure can be *expanded* toward applications with the discovery of heuristics or metrics common to a sufficiently large set of applications. Several examples illustrating this trend are listed below:

- Systems [70, 125] have been introduced that perform the functions of `stackguard` at the operating system level without requiring that applications be changed.
- Jones [65] describes an IDS which extends the short sequence anomaly detection approach of Forrest *et al* [46] by measuring library functions

invocations in place of system calls. Elbaum [42] presents a related system[5] which collects information pertaining to the path taken by a program through its function call graph in order to detect anomalous activity.

- Rooker [113] proposes that conventional operating system protection mechanisms (which operate on files or processes) be expanded to include the elements of graphical user interfaces (objects, widgets). Although Rooker identifies the NeXT system as a possible platform, no actual implementation or experience with this approach has been recorded. Although brief and lacking an implementation, Rooker's proposal is noteworthy as it is one of the earlier works arguing for the expansion of security functions toward the application level.

It can be argued that most features of language-based security mechanisms [119], implemented by systems such as the JVM [80] or CLR, can be seen to extend the infrastructure security functions in this fashion. Measures ranging from the enforcement of type safety to garbage collection, which previously were the responsibility of application programmers, have been absorbed into the infrastructure provided by higher level languages.

Time of operation:	Usually before or during the attack
Location:	Infrastructure
Degree of automation:	Automatic operation
Analysis abstraction:	Infrastructure moved closer to application
Analysis complexity:	Variable, possibly complex

5.3.6 Analysis

Security components that are located within applications occur in a number of forms and may mirror infrastructure designs. Application level reference monitors, application logging systems and even application intrusion detection systems all borrow design features originally developed as part of infrastructure components.

However, vendor provided security mechanisms tend to be less well developed than their infrastructure equivalents. This is because application providers operate under significant resource constraints as well as a consequence of the inherent specialisation of applications which removes the

[5]The work describes an instrumented operating system kernel. However, the technique can equally well be applied to applications.

economies of scale. Unlike a number of infrastructure security component designs, the costs associated with the construction of substantial application-specific security measures cannot be amortised over a large number of systems.

It is this issue of cost which makes sophisticated application-resident security components rare (although some have been constructed, by parties other than the application provider, an example being DEMIDS [34]—an IDS analysing the logs of a relational database). In order to reduce these costs, a number of automated systems (including both standalone tools or infrastructure enhancements) have been developed which can operate without the direct involvement of the application developer.

Useful as these automatic or generated mechanisms may be, a central argument of this thesis is that the construction and management of security components is unlikely to be automated completely for as as long as applications themselves are built by humans: An application can be thought to be the instantiation of an abstract process defined in terms of a domain-specific model. Flaws may be introduced if the process or domain model, as understood by the application designers, is inaccurate or if the mapping from process to instantiation is incorrect.

For most conventional applications neither the model nor process are defined formally and may not even have been articulated at all. Thus this information is inaccessible to the automated tools that inspect or augment applications.[6] This leaves only the flawed implementation as source of information. It is generally insufficient to establish if the domain model is sound[7]. Thus automated tools are largely limited to inspecting applications for inconsistencies or implementation constructs known to provide attackers with opportunities to assume control over the application. For example `stackguard` may block the exploitation of an unchecked `strcpy`, `flawfinder` will recommend replacing `strcpy` with `strcpy`, while `java` disables direct pointer manipulation entirely.

However, where domain knowledge is required to identify flaws or failures, completely automated systems are insufficient. Arguments supporting this view can be found in the work of Sielken [122], primarily a case study of the security requirements of two application domains—an electronic toll collection system and a health record management system. The domain constraints identified by Sielken show that it is difficult to transfer application-specific security concerns to infrastructure components, and are used to argue for

[6] Even if such information were available, humans would have to provide reference data to establish whether a given domain model is accurate.

[7] A case of the GIGO principle.

the construction of application specific intrusion detection systems, although Sielken's dissertation does not present an actual implementation of such a system.

The need for domain knowledge to perform effective intrusion detection can also be considered an elaboration of Pu *et al*'s [105] observation which distinguishes between *hardwired* and *contextual* flaws, where the *"first kind of attack is machine code dependent, for example, a virus or worm taking advantage of raw binary representation of programs and data ... [while] ... the second kind of attack is a more subtle one, from programs that execute legal kernel calls but somehow performing functions outside the original intentions. For example, the worm program written by R.T. Morris, Jr. uses the debugging feature of sendmail."*

It is the difficulty of equipping entirely automated and general purpose security components with sufficient domain knowledge (or context, using the terminology of Pu) which suggests that human involvement is likely to remain necessary for the protection of trusted applications.

5.4 Summary

This chapter has placed a number of security components in the classification space. For infrastructure security components it noted a trend towards more complex systems which aim to track trusted applications. Usually this approach involves security specialists constructing systems that duplicate significant application abstractions at lower levels. These shadowed application functions (protocol parsers, file format handlers, application logic) are then used to regulate the application level exchanges without relying on the guarded system.

As noted in Section 5.2, such an approach is awkward and vulnerable to desynchronisation but is deemed necessary, given that many deployed applications contain significant security flaws because application providers are only able to allocate limited resources toward security.

Automatically adding additional security components to such flawed applications is an incomplete solution. While the components may detect inconsistencies or undesirable implementation constructs, they lack the context to detect flaws which relate to inaccuracies in the application domain model.

This suggests that until machine learning systems are perfected (an AI-hard problem) human involvement will be needed in the protection of trusted applications. This makes the problem of efficiently distributing the human analysis effort among application providers, owners and security specialists interesting.

Chapter 6

Design

6.1 Introduction

The analysis undertaken in the previous chapters argues that security components operating at the infrastructure level are vulnerable to desynchronisation when used to protect trusted applications, while security components inserted automatically into applications lack the domain knowledge to be a complete defence against contextual flaws.

These weaknesses motivate the introduction of an alternative approach which aims to involve the application provider in the task of defending against security problems that are only discovered once the application has been deployed. This chapter describes the design of this component and orientates it in the classification space.

6.2 Approach

The approach used to design the security component can be contextualised by considering the degree of human involvement (the subject of the third classification dimension, see Section 4.4) in the protection of a trusted application.

The conventional strategy is to relieve the application provider of as many security responsibilities as possible and transfer these to infrastructure components. This approach is appropriate when guarding trusting applications and the only reasonable choice when applications are assumed to be malicious. Mechanisms to contain the latter have received considerable attention as part of Java and related mobile code security efforts. Examples include the work of Acharya [3] and Welch [136].

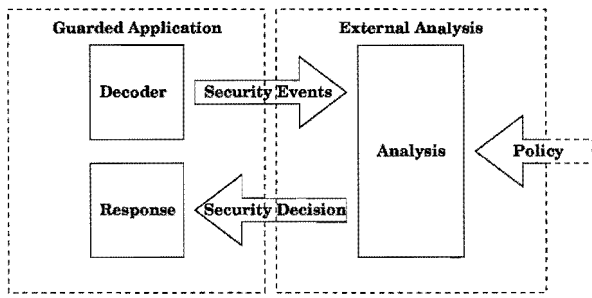


Figure 6.1: The Proposed Design mapped to the Security Component Model

However, as argued in previous chapters, this approach presents difficulties when applied to the protection of trusted applications. In particular, it results in security experts re-implementing substantial application functions at lower infrastructure levels—an expensive and, arguably, wasteful undertaking.

This chapter describes an alternative security component design requiring some co-operation from those involved in the construction of applications for the purpose of defending against contextual flaws which are only discovered after the application has been deployed. Such a component would be an intermediate, as far as the demands made of the application provider are concerned, between designs which assume application programmers to be malicious and those which demand flawless implementations.

It can be argued that this approach has not yet been explored fully, under the assumption that application providers, having failed to prevent the introduction of these flaws in the first place, are either insufficiently interested or too untrustworthy to be included in efforts to mitigate their impact.

This thesis claims that this assumption is too strong. Instead it argues that the construction of nontrivial trusted applications is simply very difficult and that applications are likely to contain flaws even if efforts were made to exclude them. A similar position is held by Cowan *et al* [36] who note that *"commercial software chronically has bugs, many with security vulnerability implications. Tempting as it may be to hypothesize that this is because the vendors are lazy or stupid, this is not the case. Commercial software chronically has bugs for the dual-reason that correctness is hard, and correctness does not sell software."*

In terms of the security component model, the alternative security component described in this chapter requires that application programmers incorporate event collection and response facilities into applications but leave

analysis to a system constructed by a security specialist, as shown in Figure 6.1. This introduces an interface between application providers on the one hand and security specialists and system administrators on the other. The purpose of this interface is to allow the former to provide the latter with the means to block the exploitation of flaws not known when the application is built, using a mechanism less disruptive and expensive than the rushed application of untested patches or hotfixes.

In this context the definitions of Elbaum and Munson [42] are helpful: *"It is a most unfortunate accident of most software design efforts that there are really two distinct set of operations. On the one hand, there is a set of explicit operations O_E. These are the intended operations that appear in the Software Requirements Specification documents. On the other hand, there is also a set of implicit operations, O_I, that represent unadvertised features of the software that have been implemented through designer carelessness or ignorance. These are not documented, nor well known except by a group of knowledgeable and/or patient system specialists, called hackers."*

Restated, it is the problem that it is almost impossible to construct a nontrivial application whose O_I set is empty that motivates the introduction of a security interface designed to allow system administrators, security specialists or even machine learning systems to monitor and adjust applications. A security reconfiguration can be initiated for a number of reasons:

- At time of installation it is considered good practice to disable unneeded application functionality. As noted by Lindqvist [82], this is particularly important in the case of COTS systems which are often marketed on the basis of the size of their feature sets. This process can be seen to reduce O_E in the hope of proportionally reducing O_I.

- The publication of a vulnerability alert (describing an element $o_i \in O_I$) marks the start of the largest phase of the window of vulnerability [120] which is only reduced once a fix or workaround becomes available. A number of reasons may delay this: Embedded applications may be difficult to service, while commercial applications depend on a single vendor (who may no longer exist or be unwilling to support the vulnerable system) to provide a remedy. Even open-source systems may not be repaired immediately, as a fix may only be available for the most recent revision (fix in CVS), with both the fix and revision being poorly tested, whereas a particular organisation may have installed a more reliable, older and possibly customised version. In these situations it is beneficial to disable the flawed application subcomponents selectively. Recommendations to this effect (workarounds) are often

included in advisories, but these are often coarse or require significant administrative effort. Using the outlined interface, a security expert (possibly even the author of the advisory) could specify the equivalent of a signature that would disable the flawed functions with only limited impact on the overall application (ie disallow o_i while retaining a reasonable subset of O_E). This would be a finer-grained intervention than the one proposed in Riordan *et al* [111] which terminates flawed applications completely.

- As part of an integrated intrusion response it may be desirable to limit application functionality in response to suspicious or unusual activity. At the infrastructure level such an automated response may be relatively coarse or easily misdirected by the attacker to damage other systems. This problem is related to the issue of desynchronisation. A strongly integrated application level response is less vulnerable to such an attack, since the path from collection to response is shorter, while a fine-grained response may reduce the costs associated with responding in error. The latter is a particularly significant concern in situations where anomaly detection systems are used in the analysis, since their false positive rates tend to be substantial [9]. Consider the case of an anomaly detection system flagging a telnet option as unusual. Blocking this option at the infrastructure level by discarding packets is likely to suspend a session indefinitely; in contrast at the application level it is possible to decline to process this option but otherwise continue.[1]

In effect, the design outlined here is an extension of the concept of defensive programming. The application programmer can use the API to request the equivalent of a second opinion whenever an action is to be taken which has security implications.

6.3 Analysis Abstractions

The previous section proposed a security component design intended to allow application providers to report security related information. This section examines the abstractions which can be used describe the reported security events. This relates the the design to dimension 4 (Section 4.5) of the classification space, which has been divided into the infrastructure, application and security abstraction categories. The tradeoffs associated with these categories are described below.

[1]Similar considerations apply to cryptographic systems where cypher types and strengths may be negotiated at the application level.

6.3.1 Tradeoffs

The three approaches can be seen to distribute the effort between application provider and security analyst in different ways.

- Reporting security events in terms of infrastructure abstractions (for example, by recording the system calls issued) requires little effort on the part of the application provider. Indeed, it is usually possible to perform this task automatically by extending compilers, rewriting executables or instrumenting the infrastructure. As noted in previous chapters, the tradeoff of this approach is that the runtime analysis is complex—it requires the duplication of application logic and is vulnerable to desynchronisation.

- Emitting events in terms of application abstractions requires the cooperation of the application provider. Whereas this approach necessitates the judicious insertion of instrumentation hooks, it does not demand that the application provider perform a translation from the native domain (for example, queries or transactions in the case of a database) to an alternative form. The corresponding analysis requires application domain knowledge and relies on the application to supply accurate information.

- Security abstractions require that the application provider map domain abstractions to ones defined in terms of a security model. Such efforts include the labelling of data, the identification of security subjects and objects or the assessment of risks. These tasks are relatively expensive but simplify the runtime security effort.

This tradeoff is illustrated in Figure 6.2. The basis for the introduction of the security component design is the conjecture that an isosecurity curve is the product of both development and runtime efforts, making the most efficient tradeoff one which involves both equally. This is supported by the observation that the extremes of this curve tend to infinity. In particular, it seems almost impossible to construct perfectly secure trusted applications (which require no security updates and thus zero runtime effort), while this thesis has argued that it is equally difficult to construct an infrastructure which eliminates the need for trusted applications entirely.

6.3.2 Approach

In order to explore this tradeoff, particularly the region spanning the application and security categories, the interface described in this chapter is

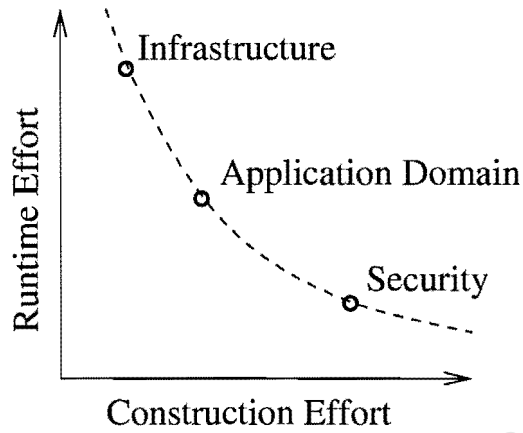


Figure 6.2: Distribution of Human Effort for Abstraction Categories

designed to allow activity to be reported in terms of multiple abstractions.

This can be achieved by using a nonprescriptive or semi-structured [2] interface at lower levels, which can accommodate several more structured frameworks at higher levels, while at the same time allowing application providers to describe activity in terms of their native abstractions with comparatively little effort.

A range of abstractions is considered. This range is depicted in Figure 6.3, which provides an approximate position of selected frameworks in terms of this tradeoff.

Most of the approaches considered are summarised briefly in this section (related frameworks have been reviewed in Section 4.5, while Appendix A.4 documents the implemented naming schemes). The exception is the functionality labelling model which is described in more detail, as it deemed to represent a particularly interesting tradeoff between implementation and deployment effort.

Native Application Domain Abstractions

This approach leaves it to an application provider to define the meaning and structure of security events. Thus subsequent runtime analysis requires application-specific knowledge.

Standardised Domain Abstractions

Events belonging to this type are defined using a domain standard. A commonly-used example is the Common Log Format (CLF) [89] which describes the requests serviced by HTTP servers and related systems. The

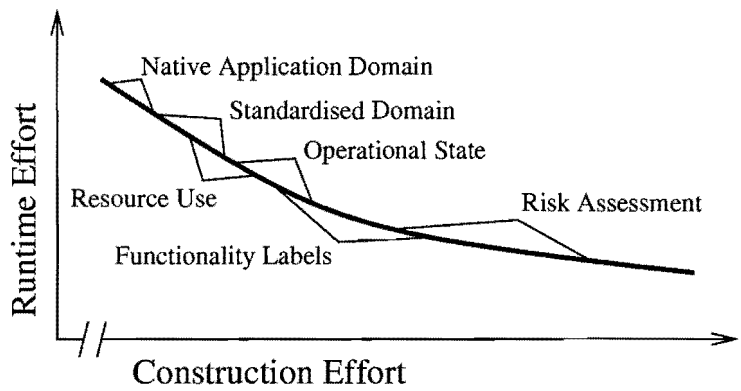


Figure 6.3: Approximate Ranges of Considered Abstractions

models which underlie SNMP [29] MIBs can also be seen as belonging to this category, although these usually pertain to infrastructure abstractions.[2]

Such an approach can be thought of as introducing a reference model for a particular domain which allows the applications operating within it to generate security events with a common structure and meaning. Often such an approach is based on an existing application protocol or similar standard (in the case of CLF, this point of reference is HTTP).

Compared to the previous category, the runtime analysis no longer requires knowledge of a particular application. This reduces the effort and makes automated analysis tools viable if the application domain is sufficiently popular.

Access Control Abstractions

In its simplest form an access control abstraction requires the application provider to describe activity in terms of subjects requesting various types of access to objects. This may be augmented by also supplying information such as group, role, domain or clearance. This abstraction is central to most operating system security components (access control, as well as intrusion detection [40]) and has similar uses within applications which introduce new subjects and objects (*eg* transactions and tables in database applications, script fragments and frames in web browsers). Recent work towards articulating and standardising this abstraction for federated web environments can be found in the SAML [54] and WS-Policy [39] specifications.

[2]Although application MIBs do exist [59, 68].

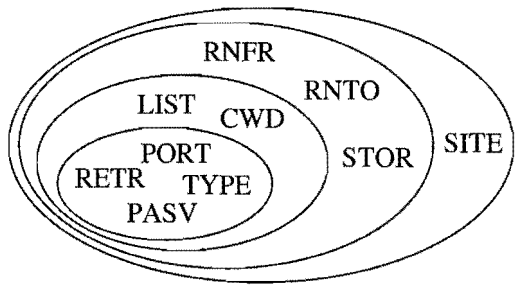


Figure 6.4: Possible FTP Server Functionality Hierarchy

Resource Use Abstractions

Events of this type describe the resources held or released by an application. Although conventionally used for accounting purposes, where known entities (often the same subjects in the above access control abstraction) are expected to pay for resources consumed, it is also useful to detect or defend against denial of service attacks, even when the resource consumers are difficult to identify. Contemporary work in this field is described by Qie *et al* [106].

Operational Abstractions

The security events of this category are related to system or network management abstractions. Central to these approaches is the notion of an overall system state, with security events generated on state changes. Examples of such transitions include an application start, suspension, reconfiguration or termination.

Functionality Labels

Security events belonging to this category describe the importance of application functions or tasks, and their associated dependencies, as opposed to labelling subjects or objects, as would be the case in a conventional multilevel or domain/type access control system.

Consider an ftp server which implements a number of application protocol commands: Some of these commands are essential if the ftp server is to perform its basic function of making files available to remote users, while others are only used infrequently. If the provider labels each of these commands accordingly, administrators or automated response subcomponents are able to initiate an orderly reduction of application functionality without requiring a detailed understanding of the application domain.

This is illustrated in Figure 6.4. The commands of the smallest set allow for the blind retrieval of files,[3] while the next largest set is sufficient to browse and download material—enough functionality to implement a read-only anonymous ftp site. The larger function sets permit modifications and access to site-specific extensions.

Such a label (which in its simplest form can expressed as a set of levels, although lattice representations are also feasible) encodes application domain knowledge, making it accessible to generic analysis. Consider, for instance, a conventional anomaly detection system that monitors the stream of ftp commands for unusual activity whose recent history includes neither `PASV` nor `SITE` commands: On encountering either, the system is likely to trigger an alert and, in the case of an inlined system, block the command. Blocking the `SITE` command may be a reasonable response, but disabling the `PASV` command is likely to be disruptive—this denies all access to clients behind restrictive firewalls. In such a situation a label indicating that the latter command provides core functionality may modulate the response.

Functionality labelling has a development cost, but it may be attractive to application providers who on the one hand wish to provide a large feature set, but also want to make it easy for administrators or automated systems to secure a system without requiring the latter to understand and adjust a large set of configuration options. A number of existing applications implement what could be considered adhoc labelling in the form of interactive configuration dialogs. These dialogs summarise or aggregate security related settings, but it can be argued that this could developed further. Such extensions would make it possible to perform more sophisticated security analysis on a per event basis, instead of fixing settings at deployment (or occasional reconfiguration) and can be used to realise the automated restrictions that have been proposed in Welz *et al* [137].

Risk Assessments

This approach involves the application provider describing pending actions in terms of the risk that they pose. Risk is described in terms of the probability of a failure occurring. The failure can be quantified as an overall cost, or as a compromise of a security objective, such as availability, confidentiality or integrity.

A risk assessment requires a substantial analysis effort by the application provider and is often a subjective exercise as past data may be scarce and of limited relevance. Thus an optimistic application provider may assign

[3]Some commands such as `USER`, `PASS`, `QUIT` have been omitted to simplify the diagram.

a lower risk rating to a particular event than a conservative one. Despite these disadvantages, such an approach is appealing because a policy defined in terms of risk thresholds can often be made sufficiently simple to be condensed to a single statement. This reduces the management demands made of application owners and administrators.

6.4 Position

This chapter has described a closely coupled security component designed to protect trusted applications and considered a number of abstractions which may be used by application providers to report security related activity to the external component.

With the time of operation and location determined by the requirements that the component intercede before it can be disabled and that desynchronisation opportunities be limited, it is possible to map the design to the security component taxonomy as given below:

Time of operation:	During the attack
Location:	Collection and response within application
Degree of automation:	Automatic operation, manual insertion
Analysis abstraction:	Application and security
Analysis complexity:	Variable

The position in the classification space relates it to other security mechanisms reviewed in Chapter 4. For example, compared to an application logging system, the proposed component operates sooner and offers a response mechanism.

Alternatively it can be viewed as a more general and developed form of `libwrap`. This library, an extension[4] of TCP wrappers [135], also externalises security analysis, while at the same time requiring the co-operation of the application provider to supply it with information and enforce the response. However, compared to the component introduced in this chapter, `libwrap` is a simpler and more specific system, managing incoming network connections only. A further difference is that its analysis phase operates on infrastructure abstractions. As noted in the previous chapters, this makes it possible to transfer most of its functions to infrastructure components, particularly network firewalls.

[4]In terms of the taxonomy, the transition from wrapper program to library shifts the collection and response location from adjacent to internal.

The outlined security component also differs from the recent[5] Embedded Sensors Project (ESP) by Zamboni [142] where a security specialist, who possesses knowledge of existing flaws, instruments systems to log attempts to exploit these flaws, even if previously repaired. Such an approach may be useful as part of a larger intrusion detection system to discover naive attackers, particularly those who indiscriminately launch scripted exploits without prior reconnaissance. However, because it requires knowledge of attacks at time of implementation, it is expensive to update as new attack descriptions become available. In terms of the security component model, both the analysis and response phases of individual sensors are fixed at time of construction. In contrast, the system described here provides integrated response facilities and allows analysis to be adjusted at runtime, both in the form of policy changes, as well as the complete substitution of analysis components.

The runtime compensation for design oversights is related to the topic of *fault-tolerance* or *survivability* [83], as these efforts also seek to reduce the impact of failures. The conventional approach involves system replication. For example, the design described by Wu *et al* [141] duplicates web servers in order to limit the impact of a server compromise. This approach requires the use of several different implementations. Consequently both development and runtime (management and resource) costs increase correspondingly. An alternative survivability measure aims to provide facilities to adjust systems in response to changing security circumstances. An example of such a system is the Security Agility Toolkit implemented by Petkac and Badger [99]. This toolkit operates by intercepting library calls made by applications and redirects them to replacements implementing additional functionality. For dynamically linked applications, the process requires neither source code modification nor recompilation.

The closely coupled security component proposed in this chapter can be viewed as an elaboration of this approach. Instead of relying solely on the infrastructure, it deliberately includes application providers in the effort to make trusted applications (using the terminology of Petkac and Badger) agile or security aware. This reduces the need to duplicate domain logic at lower levels and thus decreases the potential for desynchronisation attacks.

[5]Both the implementation described in the next chapter and the ESP were presented as short papers at the Symposium on Recent Advances in Intrusion Detection in October of 2000.

Chapter 7

IDS/A: An Application Level IDS Implementation

7.1 Introduction

The study of computer security is inherently an empirical endeavour—perhaps even an engineering discipline, rather than a science, according to Schaefer [117]. Ultimately the value of a particular security component is established during its use in protection of real-world systems. In this regard an implementation is not only needed to demonstrate that a given design can be built, but also to show that this can be done without:

1. encountering unacceptable performance losses,
2. requiring that its users incur excessive expenses or
3. introducing security problems worse than the ones being remedied.

Neither a trivial prototype nor an unreleased system may be sufficient to examine these issues.

- In the case of a prototype implementation it may not always be apparent if a performance weakness is a consequence of a substantial design problem or lacking implementation effort.
- A substantial and released implementation can be deployed. This increases the likelihood that it will be examined for security problems, since both its users as well as their attackers have an immediate motivation to understand its security characteristics.

- A published implementation is necessary to allow independent review of the results derived from its use. Failing to do so can be equated with the refusal to publish experimental methods or apparatus descriptions which is at odds with a scientific approach.

These concerns motivate the construction of a nontrivial implementation. The implemented system is named IDS/A for Intrusion Detection System / Applications since the analysis component resembles an inlined IDS, although it is also possible to view the system as an application level firewall or augmented logging system. This chapter describes the implementation, provides examples of its use and describes a number of applications which have been built or modified to make use of it. The implementation has been made available for anonymous download for over two years and has been included in a Linux distribution.

7.2 Platform

The security component design introduces an interface between applications and a security analysis component. The insertion of this interface requires source level changes. This makes a system where application sources are readily available attractive.

The technical requirements of the underlying infrastructure also relate to this interface. In particular, trusted application and security analysis component should be able to communicate (as well as execute) without interference from third parties. In terms of Loscocco *et al* [86], IDS/A requires that the infrastructure provide process isolation and trusted path[1] facilities.

The last substantial demand is that the platform is a conventional and widely used system. This ensures that both results and observations are relevant to the protection of real-world applications.

Linux meets all these requirements and was thus chosen as platform. It is popular and representative of a unix-like operating system, and its applications are generally available in source form. Linux implements reasonable process isolation using hardware memory paging and provides a number of trusted path mechanisms. These involve file paths and permissions, shared memory areas and unix domain sockets. The implementation language of the IDS/A system is C because the largest number of Linux applications are written in this language, as are most system libraries and the kernel.

[1]The term *trusted path* is commonly used to describe the communication channels between user and applications. However, it also applies to inter-application communication mechanisms.

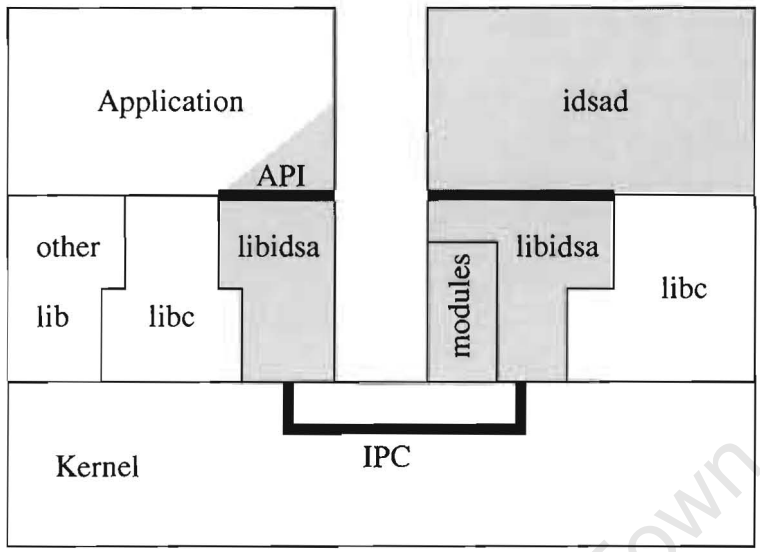


Figure 7.1: IDS/A Architecture

7.3 System Architecture

The position of the IDS/A implementation related to existing components of the implementation platform is shown in Figure 7.1 The components in gray are introduced by the IDS/A system, while the application is partially shaded to denote a modification.

Referring back to the design Figure 6.1 on page 66, the shaded region of the application contain the collection and response subsystems, while the gray components external to the application are dedicated to the analysis.

The primary interface between the application and analysis subsystem is implemented as a C API. The exchange internal to the analysis (between client IDS/A library and daemon) takes the form of a simple protocol which uses an interprocess communication mechanism provided by the kernel.

7.4 IDS/A API

The IDS/A API can be seen as the interface between development and runtime security efforts. By instrumenting an application with calls to the API, *sensors* (collection subsystems) and *actuators* (response subsystems) are embedded into the application. These can be used to modify application behaviour once deployed.

Given that application providers are unable to allocate substantial resources to security efforts, the process of adding the instrumentation to ap-

plications should be as simple as possible, since this increases the probability of it being used (see also Requirement 2 on page 77). For this reason the API has been designed to resemble an enhanced and structured version of the relatively well-known syslog logging API.

In its simplest form, three library calls are needed to report pending security-related activity for analysis:

```
IDSA_CONNECTION *idsa_open (char *name, char *credential,
                            int flags);
int              idsa_set  (IDSA_CONNECTION *c, char *n,
                            char *s, int f, unsigned ar,
                            unsigned cr, unsigned ir, ...);
int              idsa_close(IDSA_CONNECTION *c);
```

Here `idsa_open` and `idsa_close` can be seen as mandatory equivalents of the respective syslog library calls `openlog` and `closelog`. The meanings of arguments to `idsa_open` and `idsa_close` are described in more detail in the respective manual pages (see Appendix A.3). A noteworthy difference between them and their syslog equivalents is that library state is made explicit to the user via a pointer to an opaque `IDSA_CONNECTION` structure. This makes it possible to use the API in threaded applications without requiring explicit locking or atomic library internals. This should not impose a substantial burden on the application provider, since `IDSA_CONNECTION` can be regarded as the equivalent of a `FILE` handle encountered in standard stream I/O and can be hidden using a trivial wrapper (similar to the `printf` wrapper of `fprintf`).

The `idsa_set` forms the central part of the API. Its parameters provide the analysis component with information relating to pending security activity, while its return code guides the application response, indicating whether processing the pending action would lead to a security failure or not. This distinguishes it from a conventional logging function such as `syslog` which reports recently occurred activity. Thus from the perspective of the application programmer, all runtime security analysis occurs within this library call. The parameters of `idsa_set` are described in greater detail below:

`c` : A pointer to the opaque connection handle returned by `idsa_open`.

`n,s` : These two strings are used to assign a name (`n`) and namespace or scheme (`s`) to the pending security event. By using a private naming scheme, an application provider can report events defined and named in terms of a domain-specific abstraction without interfering with others.

`f` : Some security events, in particular error conditions, are neither known in advance nor can be blocked. In such cases `idsa_set` can be used in a degraded form as a structured logging interface. To indicate that it is not possible to deny an event, the caller of `idsa_set` sets the flag `f` to zero.

`ar, cr, ir` : These three parameters are used to describe the risk to *availability* (`ar`), *confidentiality* (`cr`) and *integrity* (`ir`) posed by an event. Each value encodes a cost and a confidence value—the structure of these fields will be described later. These parameters can be seen as an elaboration of the priority field of the `syslog` call. However, it is possible for an application provider to decline to provide this information by setting the confidence values of these risks to zero.[2]

ellipsis (...) : Most information is transferred to the analysis component using the variable argument list indicated by the elipsis. Unlike most other C functions taking variable arguments (such as `printf`), `idsa_set` was not designed use a format string to organise arguments, since the resultant string tends to discard structure information. Instead variable arguments are grouped in multiples of 3, each triple consisting of a label (or key) string, a type indicator and a pointer to the value. This provides a highly compact means of representing a set of named elements. Such label value approaches have previously been proposed by amongst others Bishop [20] and Abela *et al* [1] at the protocol level to structure logging information. The interface of `idsa_set` can be seen to make this available to the application programmer.

For applications reporting security information in terms of their native domain abstractions, no structure of the variable argument set is prescribed. In contrast, applications describing activity in terms of a particular security model use a predefined set of named elements which have a standardised meaning.

The return code of `idsa_set` is used to indicate whether the pending event would violate a security policy (indicated using `IDSA_L_DENY`) or is permissible (`IDSA_L_ALLOW`).

The following pages provide several code fragments which show how the API can be used. The first subsection provides a simple scenario which shows how the IDS/A API can be used to replace conventional security measures. This is followed by a short description of two alternative forms of the interface.

[2] An improvement over the priorities of `syslog` where an unknown priority is arbitrarily assumed to be a notice.

7.4.1 Conventional and IDS/A Security Instrumentation

Consider an application which provides facilities to print a document, such as a word processor, image viewer or web browser. Unix applications usually perform this task by piping the entity to be printed to the `lpr` utility. This utility is invoked using the `popen` library call. Used in a trusting environment this presents little difficulties, and at time of printing the user is often prompted to provide the parameters to `lpr` or even an arbitrary replacement. At a high level this may be implemented as follows:

```
FILE *pf;
char cmd[MAX];

strcpy(cmd, "lpr");
confirm("Print command", cmd, MAX);
if(pf = popen(cmd, "w")){
  file_export(pf, document);
  pclose(pf);
} else {
  display_error();
}
```


However, deploying such an application in a hostile environment, such as a public terminal/kiosk, presents problems. In these environments it is usually necessary to limit the user to a particular application—a restriction which can be bypassed in the above application if an attacker selects "`xterm&cat>/dev/null`" instead of "`lpr`" using a dialog of the `confirm` function.

The conventional approach to counter such problems is to enforce stricter controls at the infrastructure level, by for example running the application as an unprivileged user in a `chroot`ed environment. However, as noted previously, such controls may be coarse or duplicate application functions.[3] In such cases it is necessary to harden the application itself and do so with as little cost as possible to the application provider. Usually this is achieved by

[3]For example, the application may be relied upon to emit well-formed printing instructions, but an attacker could reconfigure or reprogram the printer. In the above this could be done using "`echo -e '%!PS...'|lpr`". This can be difficult to block at the operating system level, given that the `echo` command is a shell component of `bash`, while a shell interpreter is required for a `popen` call.

adding restrictive configuration settings and logging calls. For this purpose the above application might be rewritten as:

```
FILE *pf;
char cmd[MAX];

if(config->enable_printing){
  strcpy(cmd, "lpr");
  if(config->change_print_command){
    confirm("Print command", cmd, MAX);
  }
  if(pf = popen(cmd, "w")){
    file_export(pf, document);
    pclose(pf);
    syslog(LOG_NOTICE, "printed document \"%s\" using"
           "command \"%s\", document->title, cmd);
  } else {
    display_error();
  }
}
```

Such an approach has several limitations. The use of static configuration options requires the application provider anticipate all desired restrictions: For example, an installation may wish to allow its users to select different printers (using the option"`lpr -P printer`"), yet prevent shell access. This is not provided for in the above case and requires that the application provider add a third option which, if implemented naively, may be coded as follows:

```
strcpy(cmd, "lpr");
if(config->change_print_command){
  confirm("Print command", cmd, MAX);
} else if(config->change_printer){
  char printer[MAX];
  printer[0]='\0';
  confirm("Printer", printer, MAX);
  if(printer[0]!='\0'){
    snprintf(cmd, MAX, "lpr -P%s", printer);
    cmd[MAX-1]='\0';
  }
}
```

Unfortunately this can still be defeated if an attacker provides an escape sequence in the printer name, for example "`colour-laser;xterm&`". Countering this attack either requires substantial input sanitation (the stripping of special characters from supplied input) or the replacement of the `popen` call by lower level calls (`pipe`, `fork`, `dup` and `execlp`). The former requires a detailed knowledge of shell behaviour and is often incomplete, while the latter requires a greater implementation effort and, if used to replace `popen` completely, denies users in trusting environments useful functionality.[4]

The logging call `syslog` can also be defeated in at least two ways:

- The application can be terminated (during printing) to prevent information from being recorded.
- The attacker may assign a particularly long title to the document which ends with the sequence "`using command lpr`"". As a logged message may be no longer than 1024 bytes, this may cause a malicious print command to be discarded. Even if the title is truncated, the insertion of "`using command lpr`"" into a short title may be sufficient to evade automated log analysis tools.

The IDS/A API can be used to address some of these weaknesses. In the above example, both the code to handle the security configuration and the logging tasks can be replaced by a single call to `idsa_set` as follows:

```
FILE *pf;
char cmd[MAX];

strcpy(cmd, "lpr");
confirm("Print command", cmd, MAX);

if((idsa_set(c, "print", "application", 1, IDSA_R_DECLINE,
    "title", IDSA_T_STRING, document->title,
    "cmd", IDSA_T_STRING, cmd, NULL) == IDSA_L_ALLOW)
&& (pf = popen(command, "w"))){
  file_export(pf, document);
  pclose(pf);
} else {
  display_error();
}
```

[4]Such as the ability to print two pages onto a single sheet of paper using the command "`mpage -2|lpr`".

Here `idsa_set` is used to report a `print` event local to the `application` naming scheme. This event can be blocked (indicated by the flag being set to 1), while the provider has declined to specify risk values.[5] Two optional arguments are reported, the title of the document and the command used in printing.

In this example the IDS/A interface is used to transfer the security analysis effort encoded in the security settings to a more generic external component, while at the same time performing a more structured logging function.

7.4.2 Extended Interface Version

The library function `idsa_set` has deliberately been kept compact to reduce the demands made of the application programmer. However, the IDS/A library does also provide a more complex interface which distributes event reporting over multiple functions calls. For example, in the above example the call

```
idsa_set(c, "print", "application", 1, IDSA_R_DECLINE,
  "title", IDSA_T_STING,  document->title,
  "cmd",   IDSA_T_STRING, cmd,
  NULL);
```

is equivalent to the following fragment

```
IDSA_EVENT *evt;
evt = idsa_event(c);
if(evt){
  idsa_name(evt, "print");
  idsa_scheme(evt, "application");
  idsa_honour(evt, 1);

  idsa_add_string(evt, "title", document->title);
  idsa_add_set(evt, "cmd", IDSA_T_STRING, cmd);

  idsa_log(c, evt);
}
```

[5]The macro `IDSA_R_DECLINE` is a convenience wrapper. It expands to `IDSA_R_UNKNOWN, IDSA_R_UNKNOWN, IDSA_R_UNKNOWN`.

Here `idsa_add_string` is a convenience wrapper for `idsa_add_set` which accepts several types (integers, strings). `idsa_honour` is used to set the flag indicating whether an application is able to honour a request to block the reported event. No call to `idsa_risks` has been made as the equivalent `idsa_set` has declined to specify this information. Note that the value returned by `idsa_log` specifies if the action should be allowed or denied.[6]

The primary advantage of the long form of the single `idsa_set` call is that it can be used to generate an event whose exact number of optional parameters is not known at time of implementation.

7.4.3 Alternative Language Binding

In addition to the C API, the IDS/A implementation also provides a command-line utility `idsalog`. It can be used by applications written in languages such as PERL or for a shell interpreter in a similar manner to the C library call `idsa_set`. `idsalog` also receives a number of label value fields as command-line arguments and returns the analysis result via its exit code. The above example can thus be translated into a shell script as follows.

```
idsalog -n print -m application -f title="$title" cmd="$cmd"
```

The combination of C API and `idsa_log` make the IDS/A interface available to most Linux applications.

7.5 Analysis Abstractions

The example employed in the previous section uses a native application abstraction to describe a security event—the labels "`title`" and "`cmd`" are defined in the context of the particular document viewer or web browser. This section provides a number of examples showing how events can be reported in terms of alternative abstractions. As in the design of the IDS/A function `idsa_set`, a significant objective is to make this process as easy as possible for the application provider.

These abstractions are introduced by reserving a namespace or scheme. The labels (and possibly values) within this space are fixed in advance and assigned a meaning which maps them to entities of the particular model. These predefined elements are made available to the application provider as

[6] Also note that `idsa_log` deallocates `evt`, as documented in the manual page for `idsa_open` of Appendix A.3.

a set of C macros. Thus the task of the application provider who chooses to use a particular abstraction is to describe application activity in terms of these predefined elements.

This task can be illustrated using the example of a web server which reports an HTTP request to the IDS/A analysis system. Using a native abstraction the application provider might report this event as:

```
idsa_set(c, "request", "samplehttpd", 1, IDSA_R_DECLINE,
  "method",  IDSA_T_STRING, method,
  "url",     IDSA_T_STRING, url,
  "client",  IDSA_T_HOST,   client,
  "object",  IDSA_T_FILE,   path,
  NULL);
```

This event reports the address of the client issuing the request, the request type (`GET`, `POST`, *etc*), the url as well as the file system object to which the url has been mapped. As indicated in Figure 6.3, this approach makes relatively few demands of the application provider but requires that the runtime analysis possess application-specific knowledge.

This requirement that the analysis phase have knowledge of a particular application implementation (though not the particular application domain) can be removed by using a standardised domain abstraction. For web servers there exists an established standard, the Common Logging Format (CLF) [89], which can be used for this purpose. The IDS/A implementation provides a set of macros that define the fields of this format and which can be used to label the reported event accordingly. For example the above "`client`" can be replaced with the macro `IDSA_CLF_REMOTEHOST`.

The effort involved in mapping security activity from a native abstraction to a well-designed domain standard is often modest, as such standards often codify best current practice—what most reasonable applications would have reported anyway.

A greater translation effort is required if events are to be described in terms of specialised security abstractions—instead of a straightforward relabelling of native fields, different entities have to be identified and reported. Three examples of abstractions describing application level access control, resource use and state transitions are given in Table 7.1.

It should be noted that the examples abstract domain detail to different degrees—the access control abstraction only defines a set of actions (read, write, create, *etc*) but retains domain-specific subjects and objects. In contrast, the state abstraction does not include any domain information, because the set of states is fixed.

	Entities to Identify	Events to Report	Web Server Example	Web Server `idsa_set` Call
Access Control	Subjects and objects	Subjects accessing objects	Files requested by remote host	`idsa_set(c, "send-file", ..., IDSA_AM_SUBJECT, IDSA_T_HOST, client, IDSA_AM_OBJECT, IDSA_T_FILE, path, IDSA_AM_ACTION, IDSA_T_STRING, IDSA_AM_READ, NULL);`
Resource	Exhaustible elements	Acquisition and release of resources	Network connections made and destroyed	`idsa_set(c, "new-connection", ..., IDSA_RQ_REQUEST, IDSA_T_INT, &pending, IDSA_RQ_USED, IDSA_T_INT, &used, IDSA_RQ_TOTAL, IDSA_T_INT, &maximum, NULL);`
State	System phases	Transition between phases	Termination and restart signals received	`idsa_set(c, "shutdown", ..., IDSA_SSM, IDSA_T_STRING, IDSA_SSM_STOP, NULL);`

Table 7.1: Abstraction Examples

The implementation defines a number of other abstractions (see Appendix A.4), including ones describing risks, error behaviour and functionality restrictions (the latter having been introduced in Chapter 6). With the exception of the risk abstractions, these also describe activity by appending predefined labels and values to the variable argument list of `idsa_set` or its equivalent.

As noted previously, the risks associated with an event are described in three mandatory arguments, *viz* `ar`, `cr` and `ir`. Each value encodes a cost in the range $[-1, 1]$ and a confidence value $[0, 1]$ which allows application providers to qualify their assessment. Here a negative cost indicates a risk reduction, while a confidence value close to one indicates a high degree of certainty. The implementation provides a number of predefined risk values, including `IDSA_R_TOTAL` which denotes an almost certain and high risk and `IDSA_R_UNKNOWN` which signifies a risk assessment of which the confidence value is zero. However, it is also possible for application providers to define their own risks using the function `idsa_risk_make`. Its prototype and selected predefined risks are given below:

```
unsigned idsa_risk_make(double severity, double confidence);

#define IDSA_R_TOTAL   idsa_risk_make(1.0,0.99)
#define IDSA_R_NONE    idsa_risk_make(0.0,0.99)
#define IDSA_R_UNKNOWN idsa_risk_make(0.0,0.0)
```

Consider the hypothetical example of a database management system

reporting the deletion of a table. An application provider may choose to rate this as reducing availability, improving confidentiality and having an unknown impact on integrity. Using `idsa_set` this could be encoded as:

```
idsa_set(c, "droptable", "mydbms", 1,
         idsa_risk_make(0.2,0.3),
         idsa_risk_make(-0.5,0.5),
         IDSA_R_UNKNOWN,
        "table", IDSA_T_STRING, table,
        "transaction", IDSA_T_INT, &tid,
         NULL);
```

As indicated in Figure 6.3 on page 71, a risk assessment is a high level abstraction. It requires a substantial analysis effort on behalf of the application provider and is generally a subjective undertaking. Both confidence and costs are difficult to measure accurately—although the extremes are well defined (*eg* cost 0.0 - no effect, 1.0 - total application failure), intermediates are less distinct and are thus used to order severities rather than to quantify them precisely.

Lower level abstractions can be used as guides for such assessments. For example, a read from a unix file which is not group or world readable is likely to pose a greater risk to confidentiality than one which is. Confidentiality and integrity labels of multilevel systems could also be used for this purpose, as can the functionality labels introduced in the previous chapter.

7.6 IDS/A Analysis Components

This section describes the external analysis system of the IDS/A implementation. Its users are the parties involved in the runtime effort to secure a system and include security experts and system administrators. This contrasts with the IDS/A API whose users are the application providers.

As indicated in Figure 7.1, the analysis function is distributed over two major subsystems: A client-side library (`libidsa`) and a server process (`idsad`) which also maps `libidsa` into its address space.

The logical components of the analysis system are depicted in Figure 7.2 (a more elaborate form can be found in [138, Figure 3]). In a typical scenario an event enters the systems via a call to the API (shown at the left edge of the figure). The library adds infrastructure information (a timestamp as well as user, group and process identifiers), serialises the event and transmits it to `idsad` where it is decoded and verified. The reconstituted event is matched

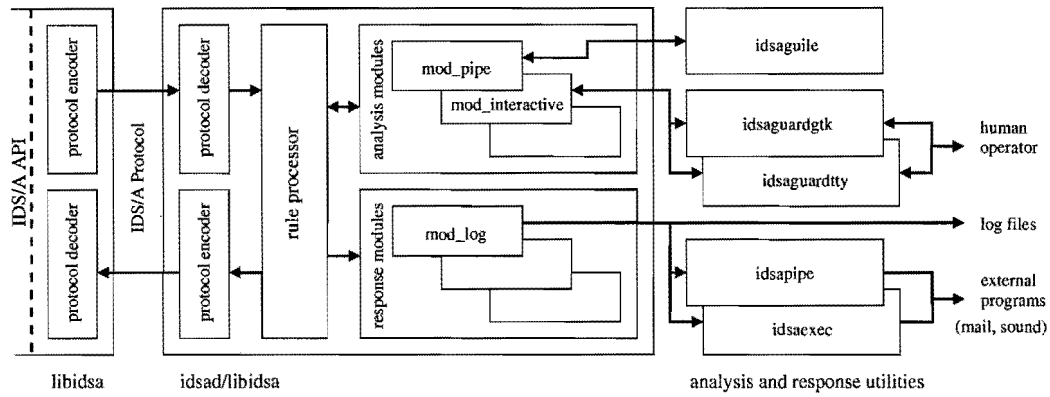


Figure 7.2: Analysis Subsystems

against a rule set whose syntax resembles an extended firewall language. Individual analysis and response functions are implemented as modules and can be extended via the module interface. The analysis result is returned to the application which processes or blocks the pending event, while the response modules initiate secondary actions, such as writing the event to permanent storage. The next sections describe these stages in greater detail, focusing on application-server communications, the rule processor and several of the modules.

7.6.1 Communications

Two characteristics define the communications between application and `idsad`: The interprocess communication mechanism and the message format or protocol.

The IDS/A implementation uses stream unix domain sockets as communication channels. Other systems using this IPC mechanism are the conventional Linux syslog subsystem, X Windowing System components, name server caches and object request brokers.

A server instance `idsad` binds one or more sockets, while applications connect to these sockets as clients. The default socket is `/var/run/idsa`. Additional sockets may be bound to support applications executing in a sandbox generated by the `chroot()` system call. Controlled facilities exist to change the default socket which makes it possible for unprivileged users to run private `idsad` instances which, for example, could bind `/home/username/.idsa`.

The format of the messages exchanged across the sockets resembles the Universal Logger Message Format [1] and an earlier one by Bishop [20]. However, unlike the logging formats, the protocol is bidirectional and includes

typing information. The first example of `idsa_set` involving a print request by a document viewer could be encoded as the following two request and response messages:[7]

```
Request {?pid:pid="1410"     uid:uid="504"     gid:gid="504"
         time:time="1043334930"     service:string="viewer"
         host:host="knoll"              name:string="print"
         scheme:string="application"        honour:flag="0"
         arisk:risk="0.000/0.000"  crisk:risk="0.000/0.000"
         irisk:risk="0.000/0.000"    title:string="unnamed"
         cmd:string="lpr -Plaser"

  Reply {!deny:flag="0"
```

Note that library call `idsa_set` inserts a number of common fields relating to the infrastructure (such as `uid`, `pid` and `time`). Consistent with the arguments in Section 5.3 this occurs automatically. Further fields (such as function name or the line number at which the `idsa_set` call is made) could also be included.

An additional binary message format has also been developed. It has been used in a number of earlier revisions atop a unix domain transport, but it is more suited to an interprocess communications mechanism involving shared memory pages.

7.6.2 Rule Processor

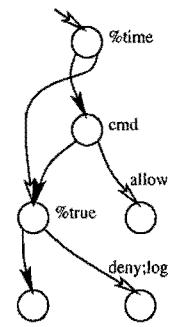
The core of the analysis system is a processor operating on a list of rules. The rule language can be thought of as an extended form of those used by network firewalls. Each rule consists of a *head* and a *body*. The rule head consists of a number of *tests* (analysis functions) which can be combined by means of conjunction, disjunction and negation operators. The rule body, which consists of a list of *actions* (response functions), is executed if its head matches the current event. An event is matched to rules in sequence and evaluation terminates at the first successful rule match unless overridden in the rule body with the `continue` keyword. The core of the grammar for this rule language is given below using a BNF notation. Optional elements are enclosed in square parentheses and nonterminals are italicised.

[7]Linebreaks have been inserted for presentation purposes. The actual protocol uses horizontal tabs as field delimiters.

rules ::=	*rule* [*rules*]	*rule* ::=	*head* : *body*
head ::=	*expr* [| *head*]	*body* ::=	*action* [; *body*]
expr ::=	*test* [& *expr*]	*action* ::=	`allow`
test ::=	(*head*)		`deny`
	! *test*		`drop`
	`label` [`:type`] [*op*] `value`		`continue`
	`%` *test-module*		[`%`] *action-module*

A simple example list of two rules is given below. The first rule, containing two tests and a single action, permits the print command "`lpr`" on weekdays, while the second rule, containing one test and two actions, denies all other events and logs them to file (note the that the first test is negated).

```
(!  %time wday sat,sun) & (cmd:string "lpr") :allow
                          \_____test_____/
\_____________head_____________/
%true :  deny ; log file /var/log/idsa/print-denied
                 \_____action_____/
\_____________body_____________/
```



At the implementation level, the list of rules is compiled into a binary directed acyclic graph which resembles a decision tree, shown adjacent to the rules. The evaluation is performed by traversing the graph. Each node encodes a single test—if the test succeeds, the true arc (outlined arrow) is taken to the next node, and any actions associated with that node are performed, otherwise the evaluation follows the false arc (solid arrow).

The rule evaluation framework is extensible and modular: Additional test and action modules can be implemented as shared objects and loaded into the system without modification to the server executable or library.

Using the long syntax, each module is introduced by a percentage sign, its name and an optional list of arguments. In the above example, the modules `time` and `true` have been used in the long form. The module `time` has been given four arguments, `true` none and `log` three.

7.6.3 IDS/A Modules

The current revision of the IDS/A implementation provides a total of 18 modules. This section describes several major module categories.

State Modules

State modules are used to store information across events. Included in this category are the modules `keep`, `timer` and `counter`. These modules are used in both rule heads and bodies. State is set in a rule body and tested in a rule head. In the following example, the `timer` module is used to record more detailed information for the minute following a suspicious event:

```
scheme:string syslog & service:string sshd &
%regex message "breakin.*attempt" :
    %timer ssh-warning 60 ; continue
%timer ssh-warning :
    log file /var/log/ssh-warnings
```

The modules can also be used to throttle activity or even shun subjects that are perceived to be hostile. However, the latter may allow attackers to initiate denial of service attacks—thus in many cases a direct response which disallows an undesired event is preferable to one which blocks subsequent activity.

Anomaly Detection Modules

The implementation provides an experimental anomaly detection module based on a class of designs described by Forrest *et al* [46]. The module can be set to examine a particular event field of successive events for unusual value sequences. Also implemented is a module to test for the occurrence of unusual characters within event fields

External Analysis Interface Modules

Two modules (`pipe` and `interactive`) provide interfaces to external analysis processes. The `pipe` module can be used to transfer selected analysis tasks to external programs, such as a shell (`sh`) or scheme interpreter (`idsaguile`). The second module forwards events to a user interface (GUI and console version implemented by `idsaguardgtk` and `idsaguardtty` respectively) for interactive analysis. While it is generally too expensive to request human analysis for every security event, it is useful to consult the operator when encountering unusual activity (in order to disambiguate between true and false positives) or events labelled as posing particularly high risks. For example, the rule list below will request permission to execute a print command not previously encountered:

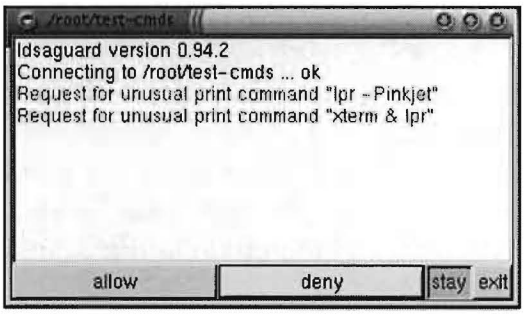


Figure 7.3: Interactive Analysis

```
name:string print & ! %keep cmd:string valid-cmds, size 64 &
%interactive /root/test-cmds 5 failclosed:
    keep cmd:string valid-cmds
name:string print & %keep cmd:string valid-cmds:
    allow
name:string print:
    log file /var/log/denied-cmds; deny
```

A screenshot of the `idsaguardgtk` utility which is used to facilitate human analysis is shown in Figure 7.3. The command line used to invoke it is given below:

```
idsaguardgtk -F$'Request for unusual print command "%{cmd}"\n'
    /root/test-cmds
```

Output Module

Output to files or other processes is handled by the module `log`. Output can be generated in a number of predefined formats, including XML and syslog, as well as used-defined ones. The latter are specified using an extended format string syntax which provides different escape sequences. Log files can be rotated automatically as soon as a size threshold is exceeded.

The implementation includes two helper utilities, `idsaexec` and `idsapipe`. `idsaexec` will spawn a command, optionally passing selected event values to the command as arguments, while `idsapipe` will pass the event stream to the process on standard input. The utilities allow for timeouts and size limits. For example they can be used to limit the number of log lines that are mailed to the administrator in one go, or close the pipe to (and so terminate) the `wall` command after a period of inactivity.

7.7 Implementation Considerations

As noted in the introduction of this chapter, a useful implementation should not introduce substantial development or performance costs, nor should it pose greater security risks than the ones it counteracts.

The emphasis on simplicity in the design of the API has sought to reduce the first concern of minimising development or deployment costs. This section describes the measures taken to secure and optimise the implementation.

7.7.1 Security Measures

The IDS/A system is implemented entirely in user space—it does not require any changes to the hosting kernel. It is noted that a modern operating system does provide applications with numerous facilities useful in a security effort.

Communications

Applications and `idsad` communicate over unix domain sockets. It is the task of the operating system to guard against the interception or modification of messages in transit. To prevent unauthorised parties from masquerading as `idsad`, the listening socket is created in the directory `/var/run` which is only writable to the superuser. In order to prevent rogue applications from masquerading as others, `idsad` uses a trusted path feature (`SO_PEERCRED`)[8] of newer unix domain socket implementations to verify the user, group and process identifier of the clients connecting to it. The same mechanism is also used to discard excessive connections from an unprivileged client who attempts to exhaust the pool of available connections. This approach provides a convenient alternative to constructing an implementation-specific authentication subsystem. The choice of local unix sockets also reduces the risk that the IDS/A interface might accidentally be exposed to remote parties, as could be the case if a transport involving TCP sockets had been used.

The choice of a specialised protocol and message format, as opposed to a generic system such as XML over HTTP, reduces the program code allocated to the communication subsystem by several orders of magnitude. In addition to improving performance, this reduction in complexity should benefit security.

[8]This capability is the primary motivation for using unix domains streams.

Privilege Minimisation and Separation

`idsad` is designed to enter a `chroot`ed environment and relinquish superuser privileges. In addition `idsad` will always lower its resource limits to disallow the creation of child processes. Thus even a catastrophic compromise of `idsad` yields only limited capabilities to an attacker (a single, unprivileged process). In this regard the IDS/A system holds an advantage over kernel resident security mechanisms where a failure may allow arbitrary operations.

Before relinquishing its superuser privileges, `idsad` can start processes such as `idsaexec` or `idsapipe`. These processes can run under different user identities. Since the interface to these is unidirectional, this may prevent an attacker who may have compromised `idsad` from undoing previous actions, such as truncating log files opened by `idsapipe` or its subprocesses.

Resource Management

An attacker may be interested in causing `idsad` to consume excessive processing time, memory or permanent storage. The following defenses are provided against these attacks:

- To counter attempts to exhaust disk space, `idsad` can automatically rotate logs once they exceed a certain size. This can be done without having to restart `idasd` or otherwise losing messages during rotation. By enabling log rotation and writing messages from different users (or messages reporting different attack types) to separate files, it is possible to enforce log quotas.

- To defend against deliberate memory exhaustion attempts as well as accidental memory leaks, `idsad` can be configured (using the `-M` option, see Appendix A.2) to preallocate all memory at initialisation.

- To prevent an attacker from consuming all available file descriptors by establishing a large number of connections, `idsad` limits the number of connections that can be held by an unprivileged user. These limits are only enforced once the pool of file descriptors nears exhaustion.

- The design of the protocol between application and `idsad` makes it difficult for an attacker to flood `idsad` with messages in the hope that some will be dropped, because the protocol uses a reliable stream transport and requires that each message be acknowledged before the next is submitted. Messages have size limits, as do individual fields. The latter reduces the potential that a value supplied by the attacker of an application interferes with other, application provided data. To ensure

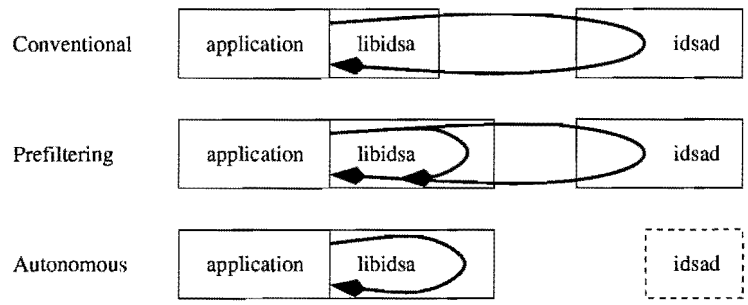


Figure 7.4: Alternative Analysis Paths

fairness, events sent by different applications are processed on a strict round-robin schedule.

- Algorithms have been selected for good worst case performance. For example, the index used in the state and anomaly detection modules (`keep` and `sad` respectively) is implemented using an AVL-tree rather than a hash table.

- The modular design makes it possible to select a particular degree of analysis complexity (see Section 4.6), ranging from the equivalent of a set of existence tests (using the default module only) to an analysis of arbitrary time and space complexity (using the guile interpreter). Particularly risk averse installations have the option of not installing high complexity analysis modules.

7.7.2 Performance Enhancements

The interprocess communications triggered by an IDS/A library call have a non-negligible impact on overall performance—the system calls used to manage the message exchange require a transfer into kernel space, while the involvement of more than one process necessitates context switches between applications.

In order to reduce these costs, the IDS/A implementation provides two facilities to perform part or all of the analysis in the client library:

- Individual applications may load a set of rules from a private configuration file entirely independently of the `idsad` process.

- The server process may push rules into the application context. This is similar to the approach used by Goldszmidt [52] for distributed systems management. These application resident rules may be used to:

- prefilter activity, submitting only activity considered hostile for analysis by the server.
- provide a fallback position, in case the server process becomes unavailable.
- transfer analysis completely into the application and disconnect from the server process.

The server pushes a rule into the application context using the module `send` to return a string containing a rule set to applications. For example, the following action instructs applications to block activity posing a high risk to availability in case the server is inaccessible.

rule head :
`send failrule:string "arisk:risk > 0.5 : deny"` ;*other action*

The rule is appended as a distinguished label value field to the protocol reply. The resulting message is shown below.

```
!deny:flag="0" failrule:string="arisk:risk > 0.5 : deny"
```

It should be noted that the rules transferred into the application context have the same syntax and meaning as the ones used by the server, because the rule processor is implemented in the library `libidsa` shared between applications and `idsad`. Furthermore, the analysis transfer is transparent to the running application—the calls to `idsa_set` remain unchanged.

However, this enhancement does change the security characteristics of system. In particular, a catastrophic failure of the client side analysis subsystem will affect the application equally severely. This is traded against the advantage of applications being able to operate independently of a central server process.

Given this change, the implementation requires that application providers explicitly enable this functionality. This is performed by setting the flag `IDSA_F_UPLOAD` in the call to `idsa_open`.

7.8 Application Examples

As part of the implementation effort a number of applications have been either constructed or modified to make use of the IDS/A API. A subset of these is listed below:[9]

[9] A number of partial instrumentations and case studies are not listed.

- `ample` [58]: a streaming audio server
- `mod_idsa`: a module for the `apache` [8] HTTP [45] server
- experimental FTP and HTTP servers as well as HTTP proxies (implemented by undergraduate students)
- ftp and pop proxies part of the application level firewall suite `fk` [71]
- `teapop` [67]: a POP3 [96] server
- `pam_idsa`: a module for the pluggable authentication module system (PAM) [115] used by programs such as `login` or `xdm`
- `linetd`: an internet superserver
- `elfingerd`: a finger server
- `idsatcpd`: a tcp wrapper replacement
- `unix2tcp`: a unix domain to TCP connection relay

The above applications cover several permutations relating to the development phase at which IDS/A calls are inserted, as well as the position of security component relative to the guarded system.

- Some applications were *designed* to use the IDS/A API, others were instrumented only after they had been constructed. Of the latter some were instrumented by *modifying* the implementation source, while in other cases calls to the IDS/A system were added using existing *extension* APIs without rebuilding the application itself.
- Referring back to the location dimension of the taxonomy (see Section 4.3), the collection and response subcomponents of an application security mechanism can be positioned either *inside* the guarded application or *adjacent* to it. The proxies and wrappers are examples of the latter.

These characteristics of the different applications are listed in Table 7.2. The last column indicates whether the application or modification has been made available to the public. It should be noted that most application types occur as components both *internal* and *adjacent* to the guarded system—for example `linetd` includes functionally which can also be provided by the external `idsatcpd` wrapper.

In addition to the above applications, the IDS/A system has also been integrated with two logging systems:

System	Instrumentation	Location	Published
`ample` streamer	modification	internal	no
`apache` HTTP server	extension	internal	yes
student HTTP server	design	internal	no
student HTTP proxy	design	adjacent	no
student FTP server	design	internal	no
`fk` FTP proxy	modification	adjacent	yes
`teapop` POP server	modification	internal	no
`fk` POP proxy	modification	adjacent	yes
`linetd` superserver	design	internal	yes
`elfingerd`	design	internal	yes
`idsatcpd` wrapper	design/extension	adjacent	yes
`unix2tcp` relay	modification	internal	no
`pam_idsa`	extension	internal	yes

Table 7.2: Selected Applications Using the IDS/A System

- `log4cpp` [12]: a hierarchical logging system. The IDS/A implementation provides a log4cpp target.
- `syslog`: the classical unix logging system. The IDS/A implementation provides local and remote syslog replacements (`idsasyslogd` and `idsarlogd`), as well as a kernel logger (`idsaklogd`).

When used as a part of a logging system, the callers of the IDS/A API only implement the collection subcomponent and defer both analysis and response to external systems.

7.9 Summary

This chapter has described a nontrivial implementation which allows the providers of trusted applications to transfer security related analysis to an external system without risk of desynchronisation.

Applications are coupled to the analysis system via a compact API. The API has been designed to make the instrumentation process simple—a sensor actuator combination can be embedded into an application with a single library call. Security events can be reported in terms of multiple abstractions, ranging from application-specific ones to high-level risk assessments.

The analysis system processing the events is modular and extensible. This permits users to select an appropriate cost-complexity tradeoff, from

cheap, stateless existence tests to the evaluation of computationally complete scripts. The analysis system can enforce conventional access control policies, perform both misuse and anomaly detection as well as request direct human supervision.

A substantial effort has been devoted to making the system useful in real-world environments—included in this effort are several measures which address security and performance concerns. In order to evaluate these measures and the overall implementation empirically, a number of trusted applications have been either constructed or modified to use the IDS/A system.

Chapter 8

Results

8.1 Introduction

This chapter presents selected results and experiences gained during the operation of the IDS/A system. The chapter is divided into four sections which describe the *instrumentation costs*, *performance impact*, *protection afforded* and the *runtime effort*.

8.2 Instrumentation Costs

The effort needed to add IDS/A support to applications depends on a number of factors, including:

- The inherent complexity of the application and its domain
- The design quality, particularly the lucidity, of the modified applications
- The ability and domain knowledge of those adding the instrumentation
- The chosen instrumentation abstraction and the level of detail selected.

These factors make it difficult to use individual results (for example, that the construction of the PAM component took a few hours or that the instrumentation of `unix2tcp` was performed within a day) as the basis for general predictions. Nevertheless, the results do convey an impression of the effort needed for representative cases.

The next sections measure this effort by reporting changes to application size and complexity. This is followed by an informal description of the effort. Metrics involving source changes were chosen over time taken because

	NBNCs	STMs	PPDs	$\frac{\Delta STM}{\Delta PPD}$
Modified Applications (source changes)				
`fkpop`	491+ 39 (7.94%)	230+ 12 (5.22%)	35+ 15	0.80
`fkftp`	955+ 20 (2.09%)	494+ 4 (0.81%)	40+ 11	0.36
`fklib`	2482+ 88 (3.55%)	1125+ 25 (2.22%)	439+ 32	0.78
`unix2tcp`	736+ 95 (12.91%)	436+ 13 (2.98%)	40+ 34	0.38
`ample`	1421+ 94 (6.62%)	661+ 27 (4.08%)	177+ 42	0.64
`teapop`	3445+391 (11.35%)	1610+ 92 (5.71%)	622+173	0.53
Extended Applications (modules or plugins)				
`apache`	+171	+ 57	+ 15	3.80
`pam`	+127	+ 64	+ 12	5.33
`tcpd`	+183	+100	+ 29	3.45

Table 8.1: Increases in Code Size (for selected applications instrumented after construction)

this approach reduces the influence of developer productivity, a factor which varies considerably.[1]

Results are given for selected applications written in the C programming language which have been *modified* or *extended* after their initial construction, because such posthoc changes make it clear what code has been added to support the instrumentation. The omission of results for applications *designed* to use the instrumentation is not considered significant because these costs tend to be lower.

8.2.1 Code Size

The size of the instrumentation has been measured using Lott's `csize` utility [87]. Both the number of nonblank, noncomment lines (NBNCs) as well as the number of statements and declarations (STMs)[2] are listed in Figure 8.1.[3]

Where source level changes have been made to an application, both the

[1] Potok *et al* [103] report a difference of *"nearly 50 times"* between most and least productive teams.

[2] Measured by counting the semicolons that are not part of literal strings.

[3] Although the wrapper `tcpd` can also be considered a new application designed to use the IDS/A interface, the fact that its sole purpose is to report connections to the IDS/A system allows it to be viewed as an extension of a network superserver such as `inetd`. Also note that `fklib` is the library which implements functions common to both `fkpop` and `fkftp`.

original size as well as the increase are given, while only the size of the plugin or module is reported for those instrumentations added by means of a runtime *extension* mechanism. The full application size has been omitted in the latter because it is difficult to define and measure:

- Applications may increase in size across revisions, yet retain the same extension interface.
- Applications of different sizes may implement the same extension interface. For example, the PAM module may be loaded by `su`, `login` or `xscreensaver`.
- Substantial application functions may themselves be implemented as optional modules. For example, most `apache` functionality is implemented in modules—the base package alone includes 33 which account for approximately half the code size.

In cases where source code has been modified, the changes are enclosed in preprocessor directives (`#ifdef USE_IDSA` and `#endif`). This makes it possible to activate the instrumentation using a compiler option. Since this inflates the NBNC count, the number of preprocessor directives (PPDs) are also listed separately. This inflation is apparent in the ratio of added statements to added preprocessor directives ($\frac{\Delta \text{STM}}{\Delta \text{PPD}}$) when comparing source modifications to runtime extensions.

8.2.2 Code Complexity

McCabe's Cyclomatic Complexity [90] was chosen to measure the complexity of adding the IDS/A instrumentation to applications. This measure quantifies the size of the application control graph. In this graph each node *"corresponds to a block of code in the program where the flow is sequential and the arcs correspond to the branches taken in the program"*. The complexity (v) can be expressed in terms of the arcs (e), nodes (n) and connected components (p—a term accounting for transfers to other modules):

$$v = e - n + 2p \tag{8.1}$$

For a structured[4] module taken as a single component ($p = 1$), McCabe shows that the complexity (v) can be derived from the number of predicates (π) contained in the module:

[4]Several of the applications also make use of unstructured `gotos`. However, since no jumps are added by the instrumentations, and existing applications use `gotos` sparingly (a single jump target for `ample`, `teapop` and `fkpop`; and four in the case of `fkftp` and `fklib`), their impact is disregarded.

	n	$\overline{v_n}+\Delta\overline{v_n}$	Δn	$\overline{v_{\Delta n}}$	$\pi+\Delta\pi$	%
Modified Applications (source changes)						
`fkpop`	26	5.04+0.19			105+ 5	(4.76%)
`fkftp`	36	7.06+0.14			218+ 5	(2.29%)
`fklib`	146	3.32	4	2.00	339+ 4	(1.18%)
`unix2tcp`	27	6.82+0.11	1	2.00	157+ 4	(2.55%)
`ample`	46	5.85+0.22	2	1.50	223+ 11	(4.93%)
`teapop`	65	8.40+0.25	3	3.67	481+ 24	(4.99%)
Extended Applications (modules or plugins)						
`apache`			9	3.33	+ 21	
`pam`			2	11.50	+ 21	
`tcpd`			1	32.00	+ 31	

Table 8.2: Increases in Cyclomatic Complexity

$$v = \pi + 1 \tag{8.2}$$

Here predicates are the language constructs that introduce branches in the control flow such as `if`, `while` or `case`. These have been counted using Cobb's `mccabe` utility and are given in Table 8.2. Separate complexities are given for modifications made to existing functions ($\overline{v_n}$ and $\Delta\overline{v_n}$ for an existing n functions) and new ones ($\overline{v_{\Delta n}}$ for Δn new functions).

For example, the unmodified `teapop` server consists of 65 functions with an average complexity of 8.4. The average increases by 0.25 after instrumentation. The instrumentation also adds 3 new functions of average complexity 3.67. Both increases are due to the addition of 24 new branching constructs to an existing 481. The distribution of the instrumentation over individual functions is shown in Figure 8.1. Each bar represents the complexity of an unmodified function, while a point indicates the complexity after modification. Points to the far left denote new functions.

8.2.3 Discussion

The previous sections measured two properties that provide an indication of the effort needed to instrument particular applications. This section describes the process in general terms and relates several observations made during the instrumentation of the above (as well as other) applications.

Referring back to Figure 6.1, the effort can be examined separately for both the collection and the response phases. The collection phase requires

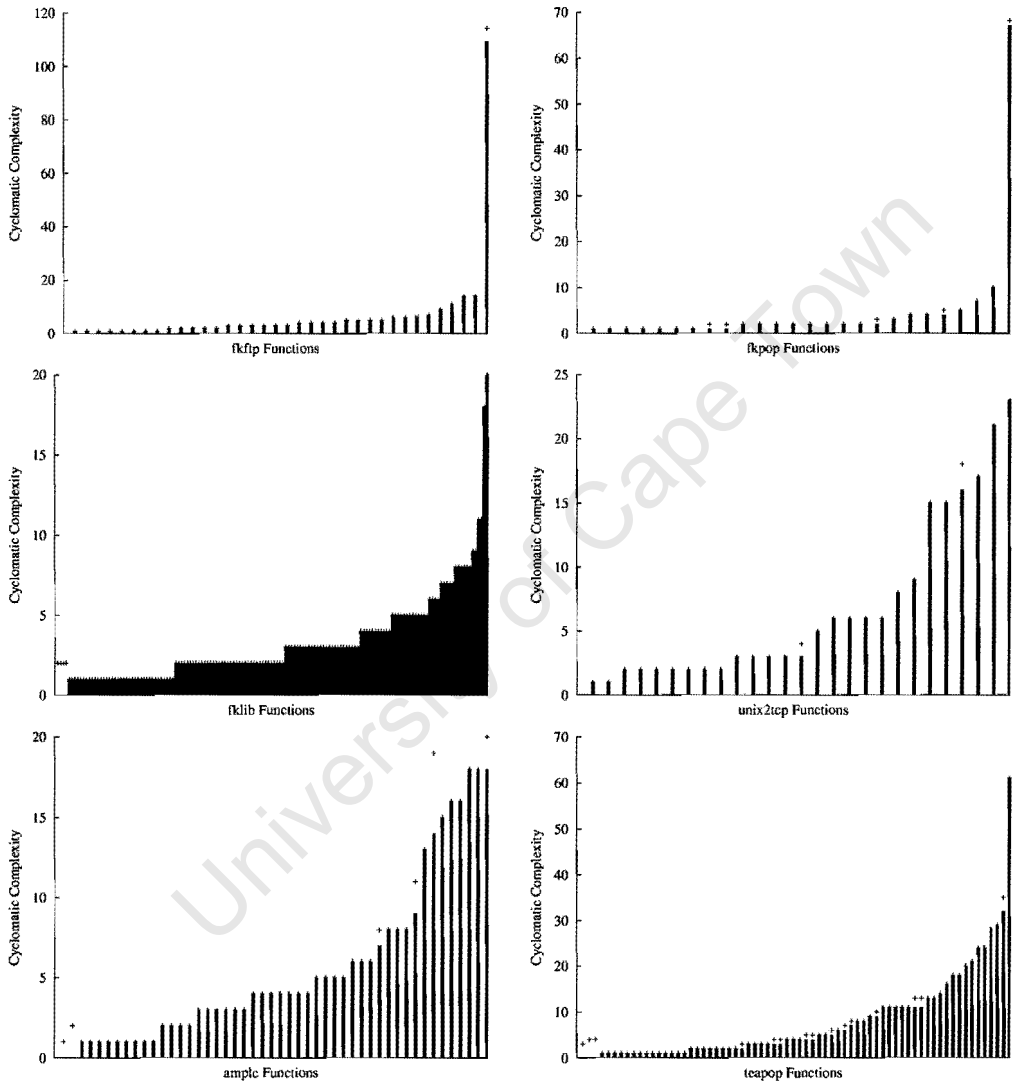


Figure 8.1: Distribution of Source Modifications per Function

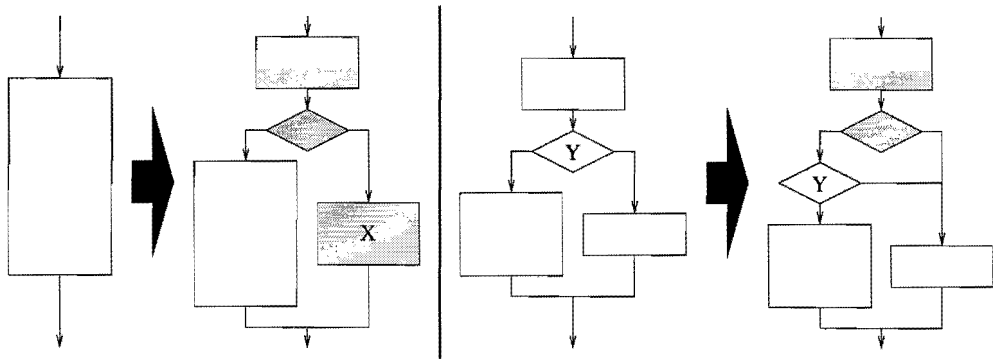


Figure 8.2: Expensive (left) and Cheap (right) Control Flow Changes

that applications be modified to supply `idsa_set` or its equivalent with information. Under unfavourable conditions an application may have to be modified to collect information which would otherwise not have been acquired, which would have been discarded or which would only have been retained in an inaccessible form. However, in most of the applications instrumented, existing data structures have contained sufficient information to populate the parameters of `idsa_set` with relative ease. This is supported (albeit indirectly) by the result that the ratio of statements to preprocessor directives in Table 8.1 indicate that the code fragments enclosed by an `#ifdef/#endif` pair is not much larger than a single statement.[5]

The response phase consists of altering the execution flow, with a deny response reducing application functionality. In suboptimal cases this task requires the introduction of a completely new decision point in the control graph of the application. However, operational experience shows that such effort is needed only infrequently as it is often possible to extend an existing branching construct. Instead of introducing a new code block (or control arc, using McCabe's terminology) to handle the response, a deny result returned by `idsa_set` is used to direct control to an existing, but more conservative, code fragment—often an error handler. The difference between the two approaches is illustrated in Figure 8.2 (X denotes the new response component). As in an earlier figure, the components in gray denote a modification.

Although these experiences support the claim that posthoc modification is not unreasonably expensive, it is worth repeating that this cost varies significantly. Using the above measurements to compute an average and variance would be of limited benefit (even if the number of samples were to be increased substantially), given that a large number of poorly quantifiable

[5]It should be noted that branching constructs themselves have not been counted as statements.

factors[6] influence the cost of instrumenting a particular application. Instead the next paragraphs describe some of the more interesting issues qualitatively.

Threads and Signals: Asynchronous programming constructs tend to increase instrumentation costs because these make it more difficult to comprehend the control flow of an application. While it is possible for the instrumentation to lock every added fragment (in case of threading) or defer/block all interruptions during its execution (in the case of signals), some subtleties tend to remain. The greater demands made by asynchronous designs can be viewed as an additional result which supports the claim by Boebert [24] that these constructs are more difficult to secure.

Side Effects: Code fragments with poorly discernible or separable side effects raise modification costs. The two cases of interest are functions used to supply `idsa_set` with data and functions used as test predicates in branching constructs altered by the instrumentation (labelled Y in Figure 8.2). A design making extensive use of accessor functions, commonly advocated as part of an object encapsulation (`getmember(object)` and the equivalent C++ `object.getmember()` instead of the unambiguous[7] `object->member`), increases these costs as an effort is needed to confirm that a read is really only a read.

Error Handling: Applications which provide superior error handling are usually easier to modify, since an existing branching predicate testing for an abnormal condition provides a suitable point for the insertion of an `idsa_set` call, as shown in the right side of Figure 8.2. Where such an insertion is made, the IDS/A analysis phase may even be used as a "fault" injection system to aid the testing of the error management components.

Instrumentation Style: A surprising factor contributing to the variation in cost is the style of the instrumentation—a *"quick-and-dirty"* change is noticeably less expensive than one which strives to follow the conventions set by the application developer. Here the term *style* not only refers to the indentation or variable naming convention but also extends to some design issues. For example, a particular application design may avoid global variables and instead transfer state in function

[6] Arguably a problem affecting the field of software metrics in general.

[7] Note that this is not necessarily true for C++. In general, the unwise redefinition of implementation language constructs also increases the instrumentation effort.

parameters—in such a case the use of a global IDS/A handle is likely to be jarring even if functionally correct.

Chosen Abstraction: Confirming the tradeoff depicted in Figure 6.2, events defined in terms of native abstractions tend to be visible directly in application procedures and data structures, while higher level abstractions need to be translated. For example, the handlers servicing individual POP protocol requests have been implemented as separate functions which are easily identified. In contrast, an access control abstraction requires that native elements be mapped to a set of subjects, objects and access modes, while the functionality labelling approach requires that actions be categorised. However, although these higher level abstractions impose a translation cost, they also guide the identification process. For example, functionality labels can be assigned on the basis of a set of questions such as *"What are the tasks performed by the applications?"*, *"Where in the application are tasks initiated?"*, *"What are the task dependencies?"* and *"What tasks are essential?"*.

8.3 Performance Impact

The previous section examined the implementation costs of adding IDS/A support to applications. This section describes the runtime overhead, in particular the increase in execution time. Like the development cost, the performance impact is influenced by numerous factors: In addition to usual infrastructure properties (operating system, compiler and hardware characteristics) affecting most performance evaluations, these results are also dependent on the particular application, its workload and instrumentation strategy. So while performance results for real applications are satisfying from the perspective of an experimentalist, they are difficult to generalise. For this reason three different approaches have been taken to measure the performance impact of adding the IDS/A instrumentation to applications:

- Detailed run times were measured for two applications
- IDS/A library calls were benchmarked on a range of platforms
- System calls issued by IDS/A functions were enumerated

The first set of measurements provides empirical results for real applications. The purpose of the second approach, the synthetic benchmark, is to remove application dependent factors and so estimate the cost of an individual IDS/A call. An application designer could use this information to

estimate the performance impact of inserting such a call at a particular location under a given application workload. While this not as convenient as a general result which claims that instrumenting applications with IDS/A calls lengthens execution time by N%, it is likely to be more useful.

Listing the system calls triggered by an IDS/A library function serves a similar purpose—it places a lower bound on the cost of an IDS/A API call in a form less dependent on a particular platform: The cost of making an IDS/A library call as measured by the synthetic benchmark is reported in elapsed seconds and is dependent on the compiler, system libraries, kernel and hardware. For a platform which is similar to the one used in the synthetic benchmark, the results should provide a reasonable indication of the expected performance impact. However, for substantially different systems, in particular newer processor architectures and alternative operating systems, such timings are of limited use. In such cases system call costs can provide an estimate, albeit coarse, of the cost of invoking IDS/A library functions.

All above measurements were made for both plain and performance optimised configurations in order to evaluate the effectiveness of shifting the computational load from a separate process into the application context as described in Subsection 7.7.2.

8.3.1 Application Benchmarks

This section provides performance figures for two applications which have been instrumented using the IDS/A API. The following five system configurations were exercised.

Configurations

Original: The unmodified application.

Simple Server: A configuration where calls to the IDS/A API inside the application are forwarded to the IDS/A server, `idsad`. The IDS/A server matches requests to a trivial rule set.

Simple Client: The same configuration as the *Simple Server*, except that the rule processing is performed in the application.

Logging Server: The same configuration as the *Simple Server* with the addition of writing each reported event to disk.

Logging Client: The same configuration as the *Simple Client* with the addition of writing each reported event to disk.

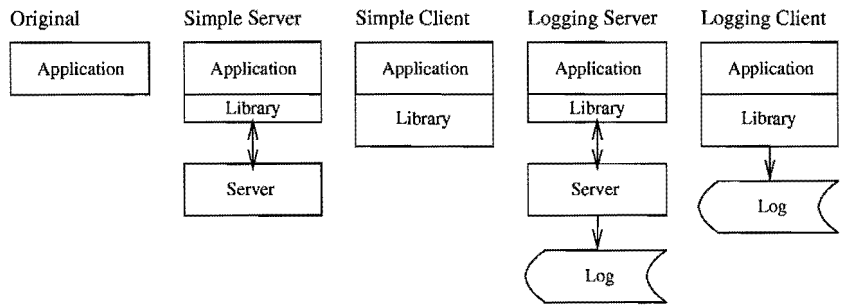


Figure 8.3: Exercised Configurations

The two simple configurations were chosen to provide an indication of the overhead of using the IDS/A API. The *Simple Client* configuration removes the interprocess communications overhead encountered in the *Simple Server* configuration.

The logging configurations have been chosen as representatives of relatively expensive tasks. They provide an indication of the cost of using the API and performing a nontrivial operation. Like the *Simple Client* configuration, the *Logging Client* configuration is intended to evaluate the performance gains of shifting tasks into the application context.

Applications

The two applications exercised are `apache` [8], a representative of an application *extended* using a plugin API, and `teapop` [67], an example of an application which has been *modified* at the source level. The applications were subjected to the following loads:

HTTP server : The web server test load consisted of 10 different web pages having an average size of approximately 60kb. Sullivan [130] found this size to be representative of documents encountered on the world wide web. For each test run 1,000 HTTP GET requests were made, distributed equally among the 10 pages. The web pages were retrieved with the `httperf` utility[93].

POP server : The mailbox server test load consisted of a 12,859,057 byte mailbox containing 968 messages with an average message size of 13,284 bytes and a standard deviation of 76,815 (the largest message occupied 1,490,421 bytes). No claim is made to the representativeness of the mailbox other than that is not synthetic. This mailbox was downloaded using the `fetchmail` [110] client.

Each configuration was exercised 25 times. In total 7,467,675,000 bytes of web pages and 1,607,382,125 bytes worth of mail were transferred.

The tests were warm, with both data and executable images previously loaded into the host buffer cache to reduce the impact of variable disk latencies.

The server and client were run on the same machine. Using the loopback network interface magnified the performance penalty of making calls to the IDS/A library—in a real scenario the network latencies and bandwidth bounds would reduce the relative impact of this penalty. Further masking can occur when an interactive client is used—both the clients used (`httperf` and `fetchmail`) are noninteractive. This issue will be expanded on in later sections.

Settings

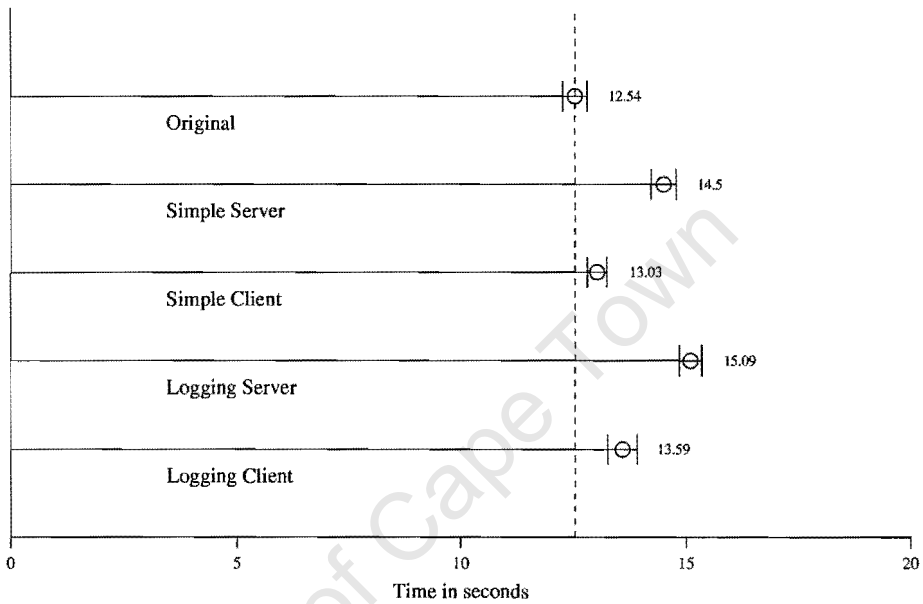
Apache Version:	1.3.14
Apache Options:	Hostname lookups disabled, logging enabled
Httperf Version:	0.8
Httperf Options:	`--hog --wset=10,1 --num-conns=1000`
Teapop Version:	0.3.3
Teapop Options:	Default, started from internet superserver
Fetchmail Version:	5.3.3
Fetchmail Options:	`--bsmtp - -p pop3 > /dev/null`
	Password read from `.fetchmailrc`
Instrumentation:	IDS/A 0.91.10
Operating System:	Linux 2.2.19
C Library:	GNU 2.1.3
Processor:	Cyrix 6x86L 120MHz
Main Memory:	64M

Results

HTTP server results are given in Table 8.3 while the POP server results are presented in Table 8.4.

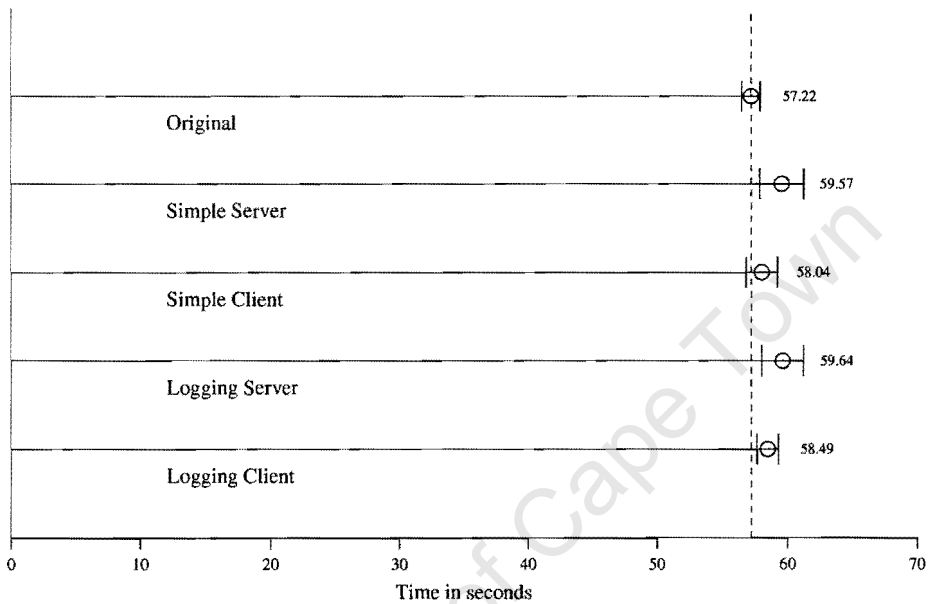
Discussion

For the unoptimised cases (executing tests inside a separate server process) the performance impact lies between 16 and 20% for the web server and 4.2 and 4.1% for the pop server.



	Mean (s)	Deviation (s)	Cost (s)	Cost (%)
Original	12.54	0.135	–	–
Simple Server	14.50	0.138	1.96	15.62
Simple Client	13.03	0.107	0.48	3.86
Logging Server	15.09	0.126	2.55	20.32
Logging Client	13.59	0.161	1.04	8.30

Table 8.3: The performance impact of the IDS/A instrumentation on the Apache HTTP server. Error bars indicate two standard deviations. Increases are given in seconds and as a percentage.



	Mean (s)	Deviation (s)	Cost (s)	Cost (%)
Original	57.22	0.347	–	–
Simple Server	59.57	0.844	2.35	4.10
Simple Client	58.04	0.602	0.81	1.42
Logging Server	59.64	0.797	2.42	4.23
Logging Client	58.49	0.409	1.27	2.22

Table 8.4: The performance impact of the IDS/A instrumentation on the Teapop POP server. Error bars indicate two standard deviations. Increases are given in seconds and as a percentage.

	Server Cost (s)	Client Cost (s)	Reduction (%)
POP simple	2.35	0.81	65.5
POP logging	2.42	1.27	47.5
HTTP simple	1.96	0.48	75.5
HTTP logging	2.55	1.04	59.2

Table 8.5: Performance Gains of Client Side Operation

In both cases the instrumentation reports application protocol level events: Each HTTP GET request results in a call to the IDS/A library, as does each POP RETR request. These library calls have similar absolute costs (all other factors being equal), regardless of whether they are made inside a web or pop server. However, as POP requests appear to be more expensive than HTTP requests, the cost of adding the IDS/A instrumentation to the pop server is a smaller fraction of the total—thus the smaller percentage.

The results illustrate why it is not possible to provide a single percentage describing the performance impact of adding IDS/A instrumentation to applications in general. The relative performance impact is specific to a particular application, instrumentation strategy, configuration and workload. This means that care has to taken not to make overly general claims based on the above results.

However, a number of mitigating factors were not considered in the above tests: Network latencies (the tests were conducted on the same host), cache misses (data and executables were preloaded) and user delays (clients were noninteractive) were excluded—these lengthen the overall execution time and thus reduce the relative impact of the IDS/A instrumentation.

It is thus not completely unjustified to suggest that under typical conditions the impact of the IDS/A instrumentation on the performance of the two applications is less than or equal to the percentages measured in the above tests.

Stronger claims can be made about the effectiveness of optimising the IDS/A system by shifting operations from a separate process (`idsad`) into a library component executing in the process space of the application. As can be seen from Table 8.5, this reduction of the performance impact is substantial.

In line with expectations the reduction for the simple cases is larger than for the expensive ones, as the interprocess communication overhead accounts for a smaller fraction of the total cost in the latter. However, even for the expensive configurations, the interprocess communication overhead is significant, and its elimination yields nontrivial gains, reducing the cost of adding

the IDS/A instrumentation to the applications to a half in the worst case to a quarter in the best.

The greater improvements for the HTTP server can be accounted for by noting that `apache` serves requests using multiple processes. These multiple processes would have competed with each other to be serviced by a single `idsad` instance—the transfer of the analysis tests to the applications parallelises this task. However, as the test was performed on a single processor system, individual web server processes still had to compete for the same processor, thus reducing the magnitude of this gain.

8.3.2 Synthetic Benchmark

The previous section quantified the relative increase in execution time of adding the instrumentation to real applications.

This section attempts to estimate the cost of making calls to the IDS/A API. This cost is measured using a simple synthetic benchmark program which makes as many calls as possible to the API.

The test used four of the five configurations described previously (the first configuration of an unmodified application is not relevant to a synthetic benchmark). The configurations were exercised on a number of different platforms, ranging from an older Pentium system clocked at 75MHz with 16Mb of main memory to an Athlon clocked at 1330MHz, with 256kb of primary cache and 256Mb of RAM.

It is noted that the performance of a given platform depends on numerous other factors besides processor type and clock speed—however, the clock speed is an indicator of the age[8] of a consumer system, and, given exponential advances in processing power over time, does summarise overall capabilities, albeit imprecisely.

Test conditions were similar to the ones for the application benchmarks. All systems were running Linux and the tests were warm.

The benchmark program was set up to issue 10,000 API calls and the elapsed times were recorded for 25 runs, resulting in a total of 1,000,000 API calls for each tested platform.

Results

The results for the trivial rule set are given in Table 8.6 and for a more expensive logging rule set in 8.7. Note that the timings are reported in seconds

[8]CPU clockspeed has a significant influence on purchasing decisions, obliging vendors to maximise this value. Consequently clockspeed is reasonably easily mapped to the age of an average desktop system.

to service 10,000 calls and that the reduction field lists the performance gain achieved by transferring the rule evaluation to the application context. Also note that the X-axes of the accompanying graphs are logarithmic.

Discussion

The figures reported for the tested systems span slightly less than two orders of magnitude for all configurations: The values computed by the benchmark program per single API call range from 0.09ms to 3.9ms for an expensive, server resident rule set, while a trivial rule set running inside the application context costs between 0.004ms and 0.31ms per call.

The cost reductions achieved by shifting the work from a separate server process into the application (here benchmark) process are even more substantial than those measured during the actual application performance tests described in the previous sections. For example, for the trivial rule set executing in the application the time taken by the synthetic benchmark can be reduced by about 90%, compared to 75% and 65% for the HTTP and POP servers respectively.

The reduction can in part be accounted for by the fact that the synthetic benchmark does not map (retrieve and possibly reformat) information from the application data structures to a representation which can be used in an IDS/A API call. These tasks may lengthen the overall execution time and thus reduce the relative significance of the interprocess communications overhead. Another, possibly even more substantial, contributing factor is that real applications are larger and thus likely to incur more processor cache misses compared to the synthetic benchmark which executes in a relatively compact loop. The absence of a load generating utility (`httperf`, `fetchmail`) in the synthetic benchmark is likely to have a similar effect.

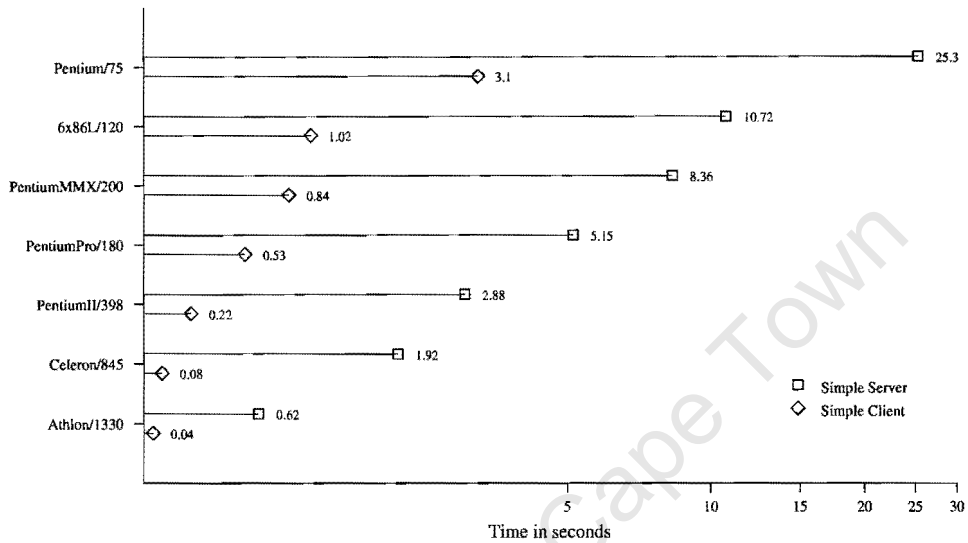
The above explanation can also account for the fact that calls to the IDS/A API in the synthetic benchmark take less time than when used in real applications. For example, it takes 10.72s to service 10,000 requests in the benchmark program for the trivial case compared to 1.96s to handle 1,000 requests in the web server.[9]

8.3.3 System Call Costs

This section presents an alternative approach to describing the performance impact of adding the IDS/A instrumentation to applications.

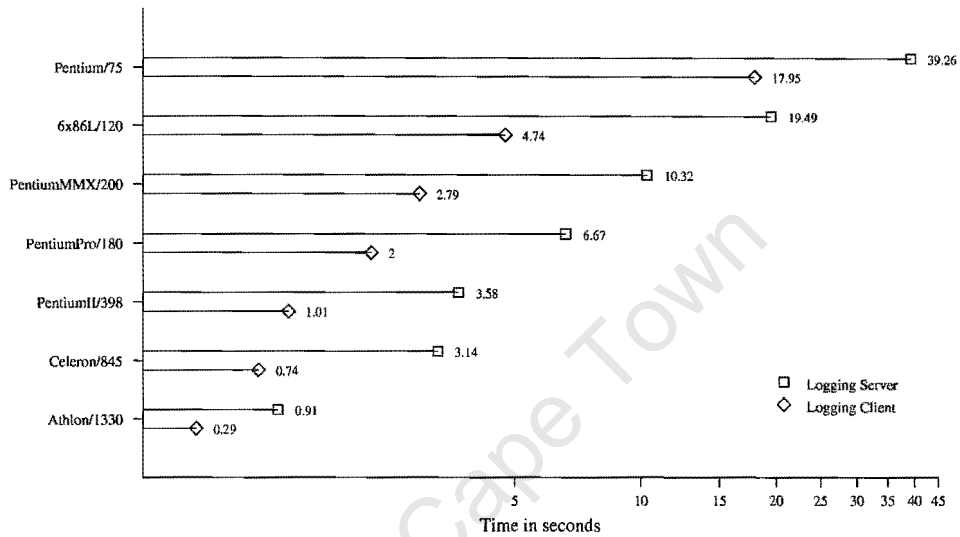
Instead of measuring elapsed execution times, this section enumerates the system calls that are made when an application invokes IDS/A library

[9]The platform used to benchmark applications is the Cyrix 6x86L.



Processor (family.model.stepping)	Clock MHz	RAM Mb	Simple Server seconds ± deviation	Simple Client seconds ± deviation (reduction)
Intel Pentium (5.2.12)	75	16	25.30 ± 0.035	3.10 ± 0.002 (87.7%)
Cyrix 6x86L (5.4.2)	120	64	10.72 ± 0.073	1.02 ± 0.013 (90.5%)
Intel PentiumMMX (5.4.3)	200	64	8.36 ± 0.127	0.84 ± 0.005 (90.0%)
Intel PentiumPro (6.1.9)	180	32	5.15 ± 0.045	0.53 ± 0.005 (89.6%)
Intel PentiumII (6.5.3)	398	128	2.88 ± 0.007	0.22 ± 0.000 (92.4%)
Intel Celeron (6.8.6)	845	256	1.92 ± 0.124	0.08 ± 0.000 (95.8%)
AMD Athlon (6.4.4)	1330	256	0.62 ± 0.003	0.04 ± 0.000 (93.6%)

Table 8.6: Time to process 10,000 events using a simple rule set measured for several generations of personal computers



Processor (family.model.stepping)	Clock MHz	RAM Mb	Logging Server seconds ± deviation	Logging Client seconds ± deviation (reduction)
Intel Pentium (5.2.12)	75	16	39.26 ± 0.346	17.95 ± 0.486 (54.3%)
Cyrix 6x86L (5.4.2)	120	64	19.49 ± 0.583	4.74 ± 0.113 (75.7%)
Intel PentiumMMX (5.4.3)	200	64	10.32 ± 0.162	2.79 ± 0.041 (73.0%)
Intel PentiumPro (6.1.9)	180	32	6.67 ± 0.100	2.00 ± 0.016 (69.9%)
Intel PentiumII (6.5.3)	398	128	3.58 ± 0.005	1.01 ± 0.002 (71.8%)
Intel Celeron (6.8.6)	845	256	3.14 ± 0.159	0.74 ± 0.059 (76.4%)
AMD Athlon (6.4.4)	1330	256	0.91 ± 0.008	0.29 ± 0.005 (68.2%)

Table 8.7: Time to process 10,000 events using an expensive rule set measured for several generations of personal computers

functions. System calls are expensive operations and constitute a significant part of the cost of an IDS/A function call.

System calls are a common subject of performance tests and are easily measured for a particular operating system and hardware configuration. A list of system calls triggered by a given IDS/A function can be used to estimate a lower bound for the cost of invoking the function on systems where such system call benchmarks are available.

Method

System call traces were recorded for the benchmark program described previously. The same four configurations of *Simple Server*, *Simple Client*, *Logging Server* and *Logging Client* were used. The IDS/A library function examined was `idsa_set`, an arbitrary choice as `idsa_scan` and `idsa_log` trigger the same system call sequence. Once-off initialisation and termination activity was not examined.

Results

The system calls issued inside `idsa_set` are listed in Table 8.8.

Discussion

The difference between a configuration which performs security related computations in a separate server process and one which performs the same task inside the application process space is clearly visible. It adds six system calls and context switches between application and server process.

The configurations that write log files can be identified by the additional `write()` system call. This system call is made within the logging module `mod_log`, however it should be noted that not all modules issue system calls when processing an event.

8.3.4 Consolidation

This section has approached the problem of quantifying the performance impact of adding the IDS/A instrumentation to applications from several angles.

The first approach measured the impact of the instrumentation on real-world applications. The application performance measurements showed that the instrumentation causes only a moderate lengthening of execution time under the particular test conditions. Unfortunately, as such performance figures are dependent on several application-specific factors, it is difficult to use

Simple Server	`time`$_C$`(NULL);`
	`send`$_C$`(server, event, ...);`
	`select`$_S$`(...);`
	`time`$_S$`(NULL);`
	`read`$_S$`(client, event, ...);`
	`write`$_S$`(client, reply, ...);`
	`recv`$_C$`(server, reply, ...);`

Simple Client	`time`$_C$`(NULL);`

Logging Server	`time`$_C$`(NULL);`
	`send`$_C$`(server, event, ...);`
	`select`$_S$`(...);`
	`time`$_S$`(NULL);`
	`read`$_S$`(client, event, ...);`
	`write`$_S$`(file, event, ...);`
	`write`$_S$`(client, reply, ...);`
	`recv`$_C$`(server, reply, ...);`

Logging Client	`time`$_C$`(NULL);` `write`$_C$`(file, event, ...);`

Table 8.8: System calls for the IDS/A API call `idsa_set()`. Subscripts indicate context, either application (C) or server (S)

them to make claims about the average cost of instrumenting applications in general. However, the results do support the assertion that there exist applications which can be instrumented using the IDS/A system at reasonable cost to performance.

Two additional approaches were used to decouple the performance of the IDS/A system from application-specific factors: Individual API call costs were both benchmarked and traced. The former provides an indication of the cost on a number of platforms, the latter a lower bound on the cost expressed as a sequence of system calls, a form reasonably independent of a particular platform.

On recent hardware (of which the Athlon-based system in Tables 8.6 and 8.7 is an example) the benchmark places the costs of making an IDS/A API call in the tens or hundreds of microseconds. For many types of applications such costs are acceptable: Applications where activity is initiated by a human and triggers only a limited number of IDS/A calls per action (*eg* most interactive applications) are unlikely to be noticeably affected by sub-millisecond increases in response time. Similar considerations apply to applications that communicate across the internet: Network latencies and variations in latency can introduce significantly larger delays than IDS/A calls.[10]

So although there is a measurable cost associated with making IDS/A calls, an effort has been made to keep this cost low enough to permit inclusion in both interactive and networked applications.

While the performance costs of adding the IDS/A instrumentation are reasonably small, there do remain some applications or usage scenarios where stringent throughput or latency requirements make the IDS/A instrumentation cost prohibitively expensive. In order to reduce the number of such cases, the optimisation of shifting the computation from a separate server into the application context was introduced. The results presented in this section show conclusively that this optimisation can reduce the execution time of an IDS/A library call, typically by a factor of two or more.

Finally it should be noted that there exist other run-time security mechanisms which, despite nontrivial performance costs, are in common use. For example, the costs associated with TCP wrappers[135] (a protection mechanism for network server applications) include the creation of a new process, parsing a configuration file and two name lookups. The popularity of systems like TCP wrappers indicates a willingness to tolerate performance losses for the sake of improved security. As systems become faster and security a

[10]Consider the example of sending a packet from New York to London: Even under optimal conditions this takes $\frac{2*5500km}{c} \approx 36ms$, about two orders of magnitude more than a typical IDS/A call on a recent system.

greater concern this willingness is unlikely to disappear. All these factors support the conclusion that the careful instrumentation of most nonrealtime applications with IDS/A API calls should not lead to unacceptable performance losses.

8.4 Security Gains

This section describes the security benefits which can be realised using the security component design proposed in this thesis. The section consists of a number of examples which are intended to provide concrete illustrations of how the proposed approach can be used to address some of the limitations identified in Section 3.3. The examples have been kept simple deliberately. This limits the amount of context needed to introduce them and makes it easier to isolate a particular issue.

8.4.1 Resistance to Desynchronisation

This example shows how the security analysis phase (used in an intrusion detection role) can be provided with more accurate information if it is coupled strongly to the applications it guards. Consider a simple reconnaissance attempt which aims to access the details of the unix administrative user via the finger protocol. `snort`, a representative of a conventional network intrusion detection system, provides the following signature to report this event:[11]

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 79
 (msg:"FINGER root query";
  flow:to_server,established; content:"root";
  reference:arachnids,376;
  classtype:attempted-recon; sid:323;  rev:4;)
```

The third line is of particular interest: It causes the signature to trigger if the string "`root`" is matched in a packet sent to the finger server. An equivalent (ignoring references) rule for an application instrumented using the IDS/A system is given below:

```
service elfingerd & name finger &
    .am-1.object:string root :
```

[11] Newlines have been inserted.

```
log file /var/log/alerts, custom "FINGER root query^J"
```

While these rules perform the same analysis task, their input data is collected by different means. The `snort` rule is supplied with data by a low-level infrastructure component, the IDS/A rule by the guarded application itself. As argued previously, the latter has the advantage of being more difficult to desynchronise at the cost of relying on the application provider to supply this information.

For example, although `snort` does attempt to track TCP connections, it was possible to discover (as part of the examination undertaken in this section) a means of desynchronising `snort` and similar systems from the applications they monitor.[12] This desynchronisation opportunity involves a complication in the handling of the TCP Urgent Flag and Pointer, a protocol feature that allows communicating parties to mark high priority data. The salient issue is that applications usually have to enable the reception of this data explicitly, by setting the `MSG_OOB` flag in the `recv()` system call or equivalent. A number of network monitors disregard this, assuming that urgent data is processed by all applications. This allows an attacker to insert misleading data into a message that reaches the weakly coupled security analysis subcomponent but not necessarily the guarded application. In the above example, a request to finger `root` can be disguised as `ro`b`ot` which does not trigger the `snort` rule. Attack payloads can be masked in a similar way.

It is not easy for a security component which only has access to low level network traffic to decode this disguised attack correctly, given that the `MSG_OOB` flag is not transmitted across the network but influences the decoding logic of the local TCP stack. In addition, the interpretation of urgent data may vary across TCP implementations and even installations—for example, the Linux kernel can be configured to follow one of two approaches.

Complications of this type provide empirical support for the claim that a weakly coupled security component is more likely to be bypassed. In this regard it is worth noting that the above example application is not even particularly complex, and that large traffic loads and encrypted communications channels provide further desynchronisation opportunities.

[12]The maintainers of `snort` have been notified of this problem, as it is not described the original desynchronisation problems list of Ptacek and Newsham [104].

8.4.2 Finer Grained Access Control

Previously the thesis stated that application level access controls tend to be relatively unsophisticated and insular—unsophisticated in that only a limited number of coarse configurations are supported, and insular in that the application access control subsystem is not easily integrated with others. The `teapop` mail retrieval server can be used to illustrate this point. The uninstrumented version of the server does not provide facilities to block access on a per host basis itself[13]—instead it relies on other systems (`tcpwrappers`, firewall rules) to perform this task. While effective, this makes it difficult to enforce finer-grained policies, such as defining host access lists on a per mailbox basis. Once instrumented with calls to the IDS/A system, it is possible to enforce such restrictions. Consider the following four rules:

```
service teapop & name login & success:flag true
    & ip4src:addr 137.158.0.0/16 :
  continue;
  keep user:string active-users, size 500, timeout 1209600

service teapop & ip4src:addr 137.158.0.0/16 :
  allow; log file /var/log/teapop/internal-use

service teapop & !name dele
    & (name connect | %keep user:string active-users) :
  allow; log file /var/log/teapop/external-reads

service teapop :
  deny; log file /var/log/teapop/disallowed
```

Taking `137.158.0.0/16` as the network local to the organisation owning the pop server, this set of rules allows all users full mail access from within the organisation (rule 2). In addition, those users who have made successful use of this service from within the organisation in the last two weeks (rule 1) are also able to retrieve, but not delete, mail from external hosts (rule 3).

This configuration offers an intermediate between a draconian policy which disallows all remote mail retrieval and one which is overly permissive—the former may cause users to circumvent controls (by installing private and

[13]This is not a deficiency of `teapop`—most pop servers only provide simple access control. Indeed `teapop` was found to be a well-written application.

poorly maintained remote access hardware and software, or forwarding all mail to external accounts) while the latter has the potential to expose a large number of dormant accounts to attack.

It is also possible to use the instrumentation to integrate policies across applications, in this example a SMTP server could be made to query the `active-users` state variable[14] to enable selective mail relaying (POP before SMTP), a task which otherwise tends to be performed using ad hoc scripts to insert hosts extracted from POP server logs into relay lists and `cron` jobs to expire them.

8.4.3 Reduction of Nonessential Trust Relationships

This section provides an example of how a more sophisticated application analysis subsystem can make it possible to reduce application reliance on the parties interacting with it. This can be seen as an implementation of the suggestion made by Gong *et al* [27] that the efforts undertaken to secure large distributed systems be shifted from the global infrastructure to individual servers or objects.

The example relates to the crawlers or robots which autonomously traverse web sites. Site owners may wish to bar robots from retrieving certain web pages if they are automatically generated, short-lived or contain sensitive information. The current procedure to achieve this, as described in [73], relies on crawlers to retrieve the list of paths not to be accessed from the file `robots.txt` stored in the document root of the web server. Such an approach is satisfactory from the perspective of the robot owner who may wish to prevent the crawler from *"getting lost"* in a dynamically generated site. It is a less attractive proposition for a web site owner as the approach depends on the *"honesty"* of the robot.

As part of the operation of the IDS/A system it was found that this trust is, at least in some instances, misplaced. By adding paths not referenced anywhere else to `robots.txt` it was found that some web crawlers retrieve (whether through incompetence or malice) the prohibited sections of a web site. This can be countered using the following two rules:

```
service apache & file:file /var/www/robots.txt :
  allow; keep ip4src:addr robots, size 1000

service apache & file:file /var/www/private.html &
    %keep ip4src:addr robots :
```

[14] Or an equivalent variable with a shorter timeout period.

```
deny; log file /var/log/bad-robots
```

The above rules can be seen to enforce a simple Chinese Wall [25] access control policy,[15] where the files `robots.txt` and `private.html` belong to the same conflict of interest class. To hinder those crawlers which do not retrieve the file `robots.txt`, the test of the first rule could be replaced with a file accessible through a link unlikely to be followed by a human, such as one embedded in a comment or displayed in a poorly visible form.

A related example involving an obligation (an AND rather than an XOR constraint) relates to the management of web server bandwidth. An undesired use may occur if a foreign server publishes a document which sources elements (typically backgrounds, icons, or other multimedia entities) hosted on the local server. Effectively this allows the foreign site to transfer a part of the bandwidth cost of hosting the document to the server under consideration. A common response to this practice is to examine the `Referer` HTTP header for nonlocal references to image elements and block these requests. As in the previous example, this relies on the co-operation of the clients to supply the correct data. This co-operation cannot always be secured, given that a number of web clients or privacy enhancing proxies omit this field or always set it to the document root of the current site.[16]

In this situation a rule set similar to the previous one can be used to remove the dependence on the client—the first rule can be modified to instruct the `%keep` module to store the addresses of hosts that have recently accessed primary documents (all html documents instead of `robots.txt`), while the second rule can be changed to supply only those addresses with supporting elements (allowing access to images instead of blocking access to `private.html`).

It should be noted that these defenses are imperfect. For example, a document author determined to include image elements from another site could perhaps embed a script that instructs browsers to load and discard a document from that site before accessing its images. In general, it is difficult to control how publicly accessible data is used by other parties. However, the above approach does increase the effort needed to circumvent policy—instead of trusting crawlers to honour the prohibitions contained in `robots.txt` or users to supply valid refer fields, the examples use a more complex analysis subsystem to check client behaviour.

[15] A Chinese Wall Security Policy is a stateful policy. It expresses *Exclusive OR constraints*—a subject may not access more than one object from the same conflict of interest class, but it is left to the subject to choose which one.

[16] For similar reasons a cookie-based approach may also not always be feasible.

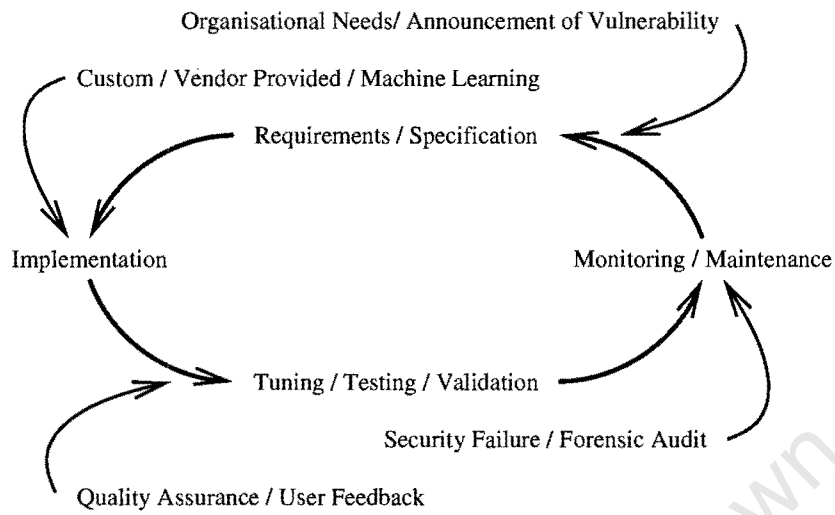


Figure 8.4: Policy Lifecycle

8.5 Runtime Effort

This section discusses the costs of operating and maintaining an IDS/A installation, in particular the effort required to formulate policy—a task comparable to maintaining a firewall or IDS ruleset. The next section decomposes policy construction into several phases and examines the different ways of distributing the effort across them. This is followed by an example illustrating how the same constraint can be enforced by different policy statements.

8.5.1 Policy Lifecycle

The policy engineering process can be represented as a typical development lifecycle or spiral, consisting of four major phases: *Specification*, *development*, *testing* and *maintenance*. This is illustrated in Figure 8.4. The next paragraphs enumerate the parties who may invest an effort at each of the phases and how these costs may be amortised.

Specification : This stage of the policy formulation process identifies the constraints which need to be enforced. Conventional access control policies are generally formulated by the organisation owning the computer infrastructure, although large-scale standardisation (for example, as part of an ERP implementation) or external legislative constraints may fix substantial parts of the policy. Where the IDS/A system is used to block the exploitation of software vulnerabilities, the equivalent of a policy requirement is implied by security advisories. For example,

CVE-2001-0473 notes that some versions of `mutt` should be prevented from processing IMAP server replies containing a format string.

Implementation : The implementation phase maps a higher-level specification to a form which can be processed by the IDS/A implementation. Arguably the most significant impact on the implementation cost is the size and computational complexity (as defined in Section 4.6) of the policy to be enforced. Implementation costs can be mitigated by *sharing* or *automating* this task. Costs can be *shared* using an opensource approach where multiple users contribute policy profiles and misuse signatures. Alternatively, or additionally, this task could be outsourced to security specialists, such as the vendors of anti-virus products (whose virus signature databases already encode a form of misuse) or managed security providers. The implementation of policy can also be *automated* by employing machine learning systems (anomaly detectors and related systems)—however, the next points will note that the latter strategy often transfers the bulk of the effort to a later stage, rather than removing it entirely.

Testing and Tuning : This phase verifies that the policies generated in the previous phase provide the desired control. Apart from conventional testing, this phase also includes the customisation of vendor provided policies and the training of machine learning systems, the latter being a task which may sometimes be as expensive as developing policies manually.

Maintenance : A set of policies enters this phase one it is used operationally. The effort needed to maintain a policy set is usually inversely related to the effort applied in the earlier stages—a poorly specified, implemented or tested set is likely to require more maintenance. Although experiences from the discipline of software engineering suggest that it is generally more cost efficient to apply the greatest effort at the earlier stages, it is probable that some organisations may not follow this approach. In such cases the `idsaguardgtk` utility may be used by security administrators to override misconfigured[17] policies manually.

As noted in the above points, the costs of formulating policy can be reduced if they are shared between organisations or transferred to domain specialists. The concept of security abstractions articulated in Chapter 6 can be seen to extend this strategy further—security abstraction make it possible

[17]Or misfiring, in the case of machine learning systems.

to simplify the formulation of policy at the cost of a greater application development effort, as indicated in Figure 6.2 on page 70.

8.5.2 Example Permutations

Given that the policy engineering costs are dependent on a number of factors which are difficult to measure (such as the competence of those specifying the policies, the quality of the instrumented application and the degree of control desired) and that costs can be distributed and transferred as described above, it is difficult to quantify the associated costs in a meaningful way. Instead this section provides an example showing how several analysis approaches (involving different degrees of complexity, human involvement and levels of abstraction) can be applied to the same security issue.

The example involves the `apache` web server which, in addition to implementing well-known HTTP methods such as `GET` and `POST`, also supports the infrequently used, but standard [45], `TRACE` method. Grossman [53] has reported that this method makes it possible to attempt a variation of a cross site scripting attack. Because `TRACE` is rarely used, it is possible to disable it completely with the rule given below:

```
service apache & method:string TRACE :
  deny ; log file /var/log/http-trace-attempts
```

The above can be seen as a simple misuse prevention signature. Such a rule could be provided as part of the security alert describing the attack or by an IDS vendor.

Alternatively, it is possible to block this activity using an anomaly-based approach, as illustrated with the following rule set:

```
service apache & %counter learning :
  allow; keep method:string valid-methods, size 32,
    file /var/state/valid-http-methods

service apache & %keep method:string valid-methods :
  allow

service apache :
  deny; log file /var/log/unusual-http-methods

service idsalog & name enter-learning-phase & uid root :
```

```
    counter learning set 1

service idsalog & name exit-learning-phase & uid root :
    counter learning set 0
```

This list of rules can be used to gather actively used HTTP methods during a learning phase which is entered when the administrative user issues the command "`idsalog -n enter-learning-phase`",[18] with new methods encountered outside this phase being rejected. This rule set can not only be used to block the `TRACE` method but also other functionality not needed by a particular site, such as the `MKCOL`, `COPY` or `MOVE` methods provided by the WebDAV [50] extension or even the `POST` method if no scripts parsing HTTP bodies are in use.

Rules of this form may be useful in a number of environments where applications provide unneeded or undesirable functionality. Early and well-known examples of such operations are the `WIZ` and `DEBUG` commands of `sendmail`, while over the last decade the `EXPN` and `VRFY` SMTP protocol operations have been disabled at a large number of sites, as they are deemed to yield too much information to potential attackers.[19] Similar considerations apply to ftp, where a number of problems have been discovered in implementations of the `SITE` command. These include CVE-1999-0880, CVE-1999-0955 and CVE-2000-0040 and have contributed to the omission of this operation in a number of recent implementations.

In cases where the set of operations (or the set of values for a particular operation parameter) is too large to be enumerated, it is possible to replace the `%keep` module with an alternative such as `%constrain` which examines a field for unusual characters at unusual positions. This may be useful to block a number of input validation flaws—attempts to take advantage of quoting deficiencies may be detected by the occurrence of escape and other magic characters, similarly buffer overflows may be made more difficult by requiring that machine instructions consist of characters only encountered during normal operation. Alternatively the sequence anomaly detector `%sad` may be used in situations where a small set of operations is in frequent use, since this module examines the order in which operations occur (*inter-*, rather than *intra-event* structure) for unusual patterns.

While the latter analysis modules are useful in some situations, their

[18] Alternatively unusual commands could be added to the whitelist interactively, as shown in Figure 7.3.

[19] The senders of unsolicited commercial email can be regarded as mounting a low-grade but sustained denial of service attacks.

operation tends to be less transparent and thus more difficult to follow. In the case of a rule list involving the `%keep` module, it is relatively easy for an administrator to examine the content of a state variable to establish if a rejected event contains a field not on the whitelist. More effort is required to decode the character statistics or sequence trees maintained by `%constrain` and `%sad` respectively.

As noted in Figure 6.2, the runtime effort (required during the specification of signatures or the management of anomaly detection modules) can be transferred to the construction phase if the developer describes events in terms of higher level abstractions. For example, the various HTTP methods could be assigned functionality labels, with only `GET` and `HEAD` marked as essential. This would make it possible to configure the web server to offer only minimal functionality with the following rule:

```
service apache & .fnl-1.level:integer > 0 :
  deny
```

Given that this rule operates at a higher level of abstraction, some loss of detail is to be expected. Such a tradeoff may be acceptable if policy is to be set by individuals with limited knowledge of application detail. It can also be used to mount a general and entirely autonomous, if primitive[20] response by, for example, coupling the currently available functionality level in multiple applications to a global threat counter.

8.6 Summary

This chapter has presented selected results and experiences describing the effort, performance impact, security benefit and administrative effort of *externalising* the security analysis tasks of trusted applications. The information was gathered over a period of three years during the construction and operation of a system exploring this concept. The implementation is non-trivial and has been made available to the public, making the results more credible than ones which are derived from a simulation, skeletal prototype or unpublished system. However, a real-world implementation also introduces a substantial number of external factors which are difficult to control. Thus the results should be seen as supporting an existence ($\exists$), rather than a universal ($\forall$) claim. Summarised, the results show that there exist applications that can be modified with moderate effort (an increase of below 5% in

[20] Akin to a snail retreating into its shell when prodded.

number of application decision points) and with an acceptable performance impact (typically millisecond to microsecond delays per analysis invocation) to produce an integrated security mechanism capable of fine-grained and preventive intervention which is less easily bypassed than a lower level intrusion detection system.

Chapter 9

Discussion

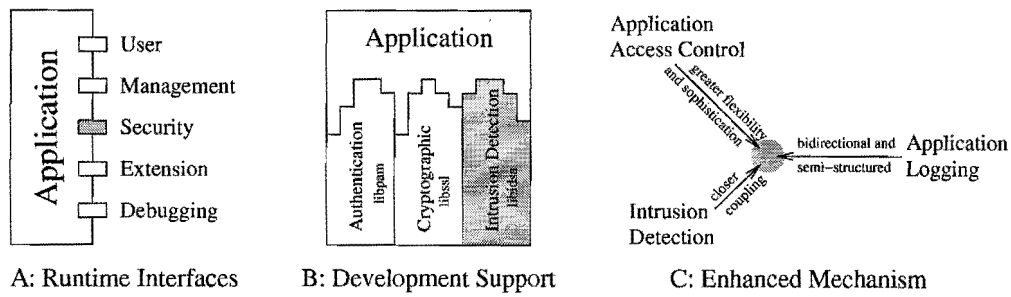
9.1 Introduction

This chapter examines the work described previously from a higher level perspective. It considers the limitations of the approach, the social factors relating to it and possible areas for future investigation.

9.2 Context

Constructing secure software is a hard problem: Over the last several decades numerous software failure modes have been discovered and described, but only limited progress has been made in eliminating them. Given the past lack of progress and the substantial investments in existing infrastructure (the latter one of several factors, according to Pike [100], which impede systems research in general), it appears overly optimistic to expect that these problems will be solved by the discovery of a revolutionary technique.[1] Hence Bellovin [15] and others argue that efforts should be directed towards improving existing systems incrementally and making provision for failures. The approach proposed in this thesis should be viewed in this context—in place of rejecting the prevailing penetrate-and-patch paradigm, it aims to refine the process by adding an interface which permits the runtime adjustment of applications for the purpose of discovering and blocking undesirable activity. This can be used to reduce the penetration possibilities by disabling unneeded functionality or expedite the patching process. Instead of developing, testing and installing a new software release which repairs a flaw or provides more detailed restrictions, the equivalent of a signature can be disseminated

[1] Although it would certainly be pleasing if a breakthrough were to be made.



A: Runtime Interfaces B: Development Support C: Enhanced Mechanism

Figure 9.1: Alternate Views of the Proposed Mechanism

which restricts application behaviour.[2] Even in cases where such restrictions are imperfect, they may be useful as interim measures while a full response is being formulated.

Viewed from the perspective of the application user or owner (Figure 9.1A), the thesis proposal can be seen to add a dedicated, bidirectional security interface to trusted applications, in addition to any existing user,[3] extension, debugging or management interfaces.

To the application developer (Figure 9.1B), the approach appears as a mechanism that allows the delegation of intrusion detection and access control-related analysis tasks. Viewed from this perspective, the IDS/A implementation extends the scope of security related APIs, which are normally used to relieve application developers of tasks such as the implementation of cryptographic envelopes or authentication subsystems. In the same way that, for example, the introduction of the pluggable authentication module (PAM) client API has increased the flexibility and sophistication of the sign-on procedures, the IDS/A API is intended to enhance the response to undesirable activity.

Related to existing security mechanisms (Figure 9.1C), the proposed approach can be viewed as an elaboration of one of three existing ones: It can be regarded as a more sophisticated and externalised application access control system, an application logging system which is more structured and includes a response pathway, or it could be seen as a closely coupled intrusion detection system which involves applications in the data collection and response phases.

[2] A more cost effective proposition, according to Fiebig [44] who is quoted as stating that system patching is significantly more expensive than a signature update.

[3] In this context any channel by which the application renders its service, not only the interface to a human user.

9.3 Limitations

The proposed approach differs from conventional deployed security components in that it relies on applications both to collect information and to enforce a response. While the previous chapters have argued that this approach is attractive, it is not without limitations. In particular, it presupposes that application developers (or other parties, in the case of open-source software) are willing to apply the instrumentation, and depends on the integrity of the applications once instrumented. Some of the social factors influencing the decision to instrument a particular application are examined in the next section. This section considers the disadvantages of relying on the application.

The primary restriction of the approach is that the security analysis component has to be invoked before an application can be exploited because the response mechanism consists of directing control flow away from the part of the application which would have yielded an advantage to the attacker.[4]

The secondary concern is the supply of data to the analysis component. Although in some cases it is possible to block a failure given only an identifier naming the pending operation, it is usually preferable to receive more information to allow a more precise response. Consider the example of an ftp server implementing the infrequently used `SITE` command. If a flaw in the command implementation is discovered, it may be feasible to disable the command outright without needing any further information. This may not be the case if a weakness is found in an essential application function, such as the handling of the ftp file retrieve command `RETR`. Here additional information such the file name requested (does it, for example, contain the substring `/../` ?) or the username (has the user previously made use of the service?) would be useful to restrict access selectively to known users who do not exploit the vulnerability.

The location and density of the points at which the external analysis system is consulted, as well as the amount and type of information reported at each point, is largely dependent on the judgement of the developer adding the instrumentation, although the higher level security abstractions may offer some guidance. For example, an access control model may assist in the task of identifying the points at which a subject acts on an object. However, imperfections are likely to remain, resulting either in failures not being blocked or coarser than necessary responses.

Despite this limitation, the proposed approach is deemed attractive given

[4]Exceptions to this restriction involve partial application failures or situations where the instrumentation is part of a system adjacent (using the definition of Figure 4.2) to the guarded application. In these cases the approach may also be used to contain an attack after a certain degree of damage has been caused.

that the applications under consideration are trusted systems. Thus their developers, while not infallible, are non-malicious with some interest in security. This makes it feasible to take advantage of developer knowledge instead of duplicating significant application logic and state at lower levels—the latter the strategy of conventional intrusion detection systems which is, as argued previously, expensive and vulnerable to desynchronisation.

The above position should not be interpreted as an outright rejection of infrastructure security mechanisms, but rather as the contention that classical infrastructure access controls are necessary but not sufficient for the protection of trusted application, and that instead of increasing security infrastructure complexity to track application behaviour, it seems more appropriate to strengthen application level security mechanisms.

9.4 Social Factors

This section examines selected nontechnical issues which may influence the proposed approach in a real world environment. The main focus is on factors which may encourage or hinder its adoption, followed by a discussion of the privacy implications.

9.4.1 Adoption

In order to persuade developers to adopt the approach, it would be useful to find a means of describing the possible benefits succinctly. For this purpose the security instrumentation process could be presented as a defensive programming technique where diligent developers, aware of the possibility that their applications may be imperfect, incorporate facilities to monitor and adjust their systems at runtime. Two analogies might be helpful in this regard:

- A medical comparison could be invoked, which portrays the interface as a means for the application (developer) to solicit a "*second opinion*" from a security expert when about to perform a potentially dangerous operation.

- The instrumentation could be described as a form of insurance which is used to mitigate the effects of flaws not discovered during testing.

Particularly the latter may assist in presenting the mechanism as part of a strategy to address the issue of retrofitted security, a problem caused by the failure to consider security implications during the implementation of the

design prototype (as it is deployed in an environment which is not hostile) and the first release (since the latter is usually constructed under severe time pressure). Consequently security measures tend only to be considered after the release of several major revisions when the publication and exploitation of security weaknesses have made the security problems too significant to ignore. Here the likes of the IDS/A API could be used by developers to flag potential security concerns during initial application development, but defer complete analysis to a later stage or transfer it to other parties entirely.

A result [134] from the field of Prospect Theory [66] indicating that humans tend to prefer a future, larger but uncertain loss over an initial, smaller but certain expense (even if the latter is statistically somewhat cheaper) may have a bearing on the discussion. Under a pessimistic interpretation this result suggests that developers are unlikely to apply the instrumentation because it constitutes an upfront cost, even if smaller than the one associated with a potential security failure. However, it is also possible to interpret the result as an argument favouring the approach, if the instrumentation cost is compared to that associated with the elimination of security flaws before deployment.[5]

If implementation effort remains a concern, it is possible to present the system as an augmented logging interface. For new applications requiring logging facilities, the augmented interface not only provides a message format more amenable to automated analysis than unstructured text, but it also includes an immediate response pathway at little additional cost.

Although these arguments may be useful in the promotion of the approach, a number of outstanding issues may have to be addressed if the mechanism described in this thesis is to be adopted on a larger scale. Amongst others, it is likely that the above work, conducted as an independent research effort, would have to be transformed into a more inclusive process leading toward ratification as a standard or other specification permitting multiple, independent implementations. The value of a champion of the approach, in the form of a well-funded or respected organisation should also not be underestimated.

9.4.2 Privacy Concerns

The proposed instrumentation can be used to collect extensive and detailed information. This provides a means to violate the privacy of an individual. To address this problem in conventional audit systems, Biskup *et al* [22]

[5]The observation that the majority of flaws remain latent over the lifecycle of a system is also relevant, provided it holds for trusted application.

suggest replacing sensitive event fields (user identifiers or network addresses) with pseudonyms that are sufficient to perform the analysis but do not identify the user. The mapping from pseudonym to real identity is only revealed if the analysis discovers a transgression. Such a facility could also be added to the IDS/A implementation. However, under certain circumstances it may be possible to achieve a similar result with a simpler mechanism: Given that the collection and analysis phases of the IDS/A system are closely integrated (in particular events are not written to permanent storage before analysis), the system could be configured to record only events which the analysis flags as suspicious.

In general, it is the process of recording user activity over a period of time, rather than the analysis of current actions, which poses the primary threat to privacy. Under utopian conditions the proposed design could be used to reduce the need for extensive records and so enhance privacy, since it is intended to enforce policy by means of disallowing undesired actions, rather than by the threat of repercussions if evidence of undesired activity is discovered in the trail of security events. Unfortunately, like most other systems aiming to detect undesirable behaviour, the closely coupled analysis is imperfect. Thus it usually remains necessary to record substantial information for human review. In these situations the publication and enforcement of reasonable data retention policies play a substantial role in the prevention of privacy violations.

9.5 Future Work

Several aspects of the work performed as part of this thesis could be elaborated. Opportunities exist to extend the implementation, to investigate the security-related analysis abstraction concept further and to develop business models around outsourced application security analysis.

9.5.1 Implementation

A substantial effort was made to create an operationally useful implementation. Thus the resulting system could be employed as an experimental platform in other investigations. For example, the system could be used to exercise an alternative anomaly detection strategy—the detection logic could be implemented as an IDS/A analysis module, allowing it to utilise existing application level data collection points. Alternatively the instrumentation could be used to alter the control flow of an application, either to test and debug infrequently used paths, or to explore a particular failure hypothesis

postulated during a failure mode and effects analysis.

Apart from using the system in a support role, it may also be possible to expand its scope along the two lines described below.

Infrastructure Extension

The implementation is primarily an application level security mechanism, but it may be feasible to extend it towards lower levels by intercepting and modifying kernel security decisions. The userspace `ptrace/strace` mechanism, but also kernel modules or direct kernel modifications, have been used for this purpose by systems such as Janus [51][6] and Medusa DS9 [143]. A similar approach could be used by the IDS/A system to handle security analysis for both a kernel and the applications it hosts.[7]

The unification of infrastructure and application security subsystems may be interesting in, for example, exploring the possibilities of combining voluntary (or internal) and mandatory (or external) application controls. However, it is also not without risk, as the merger of two previously distinct layers may reduce defensive depth.

Distributed Extension

The current implementation is designed to operate within a single host. Applications and `idsad` communicate using a channel where message authenticity, integrity and confidentiality are guaranteed by the host operating system.

Distributing the system over multiple hosts would require the replacement of the local communications subsystem (unix domain sockets) with one spanning hosts (most likely TCP/IP sockets).

A naive replacement of this form is trivial. However, as network channels tend to be less secure and reliable than those internal to a host, it would be necessary to add the following mechanisms:

- A cryptographic envelope to make the interception or injection of messages more difficult. This facility could be provided by an SSL/TLS wrapper or extension but also by the likes of IPSEC at lower levels.
- A system to deal with issues of reliability and latency. The already implemented performance enhancement of transferring of rules into local applications could be adapted for this purpose.

[6]While the original Solaris implementation uses the Solaris `/proc` tracing interface, the Linux version uses its own kernel module.

[7]An interposition wrapper in the form of a preloaded shared library to report and possibly block the `exec()` family of calls (similar to one proposed by [11]) has already been implemented. However, this can be bypassed.

The introduction of such an extension should not be particularly disruptive, since the interface to applications is defined by an API, not a transport protocol. This higher-level interface shields application developers from such changes and because the API is contained in a shared library, different implementations can be substituted without modifying the application executables.

As in the case of the infrastructure extension, it is not so much the technical feasibility of implementing the addition, but rather the mitigation of increased security risk which would require the most attention. In particular, the properties of a distributed system (unreliable inter-host communication and larger size) are likely to make such a system more accessible and more attractive to potential attackers, the above measures notwithstanding.

9.5.2 Security Abstractions

The thesis considered the distribution of the security effort over the development and deployment phases of an application lifecycle. It suggested reporting events in terms of higher level security abstractions in cases where it is desirable to transfer the bulk of the effort to the developer. Opportunities exist to investigate further abstractions and consider the utility of this approach in general.

Alternative Abstractions

It may be interesting to develop additional abstractions which can be used to capture a particular application security aspect. Other fields of endeavour could serve as inspiration and be examined for interesting analogies. An example of this approach has been taken by Moskowitz *et al* [94] who consider a computer system to provide a set of barriers checking the flow of insecurity across domains. The barriers can be encoded as nodes and the domains as edges of a graph. Given the reliability of each barrier and its connectivity, it is possible to borrow an analysis technique used to calculate the current flow through an electrical circut to compute the robustness of a system, based on the observation that barriers arranged in parallel offer less protection than those arranged in series.

An illustration of how this abstraction can be applied to inter-application security relationships is shown in Figure 9.2, and it is possible that further analogies await discovery or use in the effort to describe security features in a form less dependent on the particular application domain.

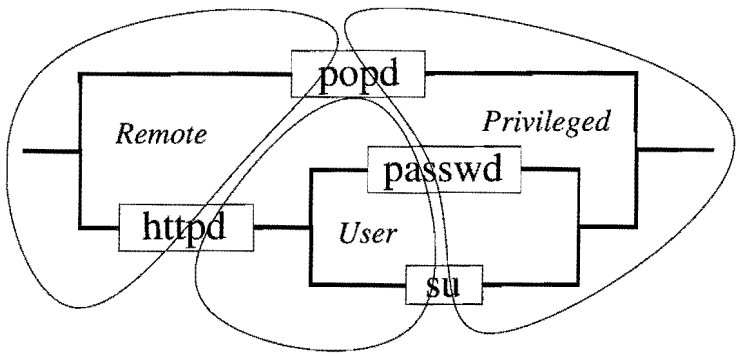


Figure 9.2: Insecurity Flow Across Trusted Applications

Elaborated Abstractions

It would be interesting to continue the investigation into how security-related information can be encoded in a form independent of the originating application. If it were feasible for application developers to describe all security implications in such a way, the need for expensive domain knowledge during analysis (either directly for the formulation of application-specific logic, or indirectly in the supervision of thus far imperfect machine learning systems) could be eliminated completely.

However, it seems unclear if such an approach is currently viable, as it is likely to require that the application be mapped to a security model of substantial complexity if all the features are to be captured. The mapping to such an elaborate abstraction may itself contain flaws hindering fully automated analysis. In addition its cost may be comparable to a formal verification process, making it unaffordable in the majority of cases.

Hence it seems more productive to develop smaller abstractions of the form described in this thesis which, while capturing the security concerns only incompletely, are easier to comprehend and thus more likely to find widespread use. If successful, these could then be combined gradually to generate more sophisticated systems, instead of introducing a complex and unproven abstraction immediately.

9.5.3 Managed Security

The introduction of a distinct application security interface may offer opportunities to the providers of security services. Apart from contractors who may be involved in adding the instrumentation as part of a security audit or review, the interface should be of greatest interest to managed security providers, as it introduces defined points at which the systems of an organi-

sation can be accessed. Managed security providers could use this interface to:

1. acquire more detailed and accurate information for analysis,
2. sell hardened configurations which limit application functionality to a subset appropriate for a hostile environment, and
3. provide security updates which resemble virus[8] or misuse detection signatures but which are able to prevent applications from processing attack payloads designed to trigger known flaws.

In general, such a dedicated interface would allow external security specialists to focus on the security analysis, rather than the data acquisition phase (with the associated costs of constructing and maintaining custom data collection devices) or even the overall management of an organisation's computing resources. In other words, the proposed interface may help delineate the role of a security specialist when compensating for application provider oversights, whether involving overly trusting or featureful applications (Point 2 in the above list) or vendors slow to repair flawed releases (Point 3). In a larger context, this could facilitate the transition of security specialists from the sale of a product to the provision of a service.

[8]Here the observation of Swimmer [132] that the intrusion detection industry resembles a less mature anti-virus industry may be of relevance. In particular, the dominant virus defense model (rapid dissemination of new virus signatures) can be interpreted as suggesting that misuse, rather than anomaly detection, is likely to remain the primary defense against intruders.

Chapter 10

Conclusion

The thesis has examined the problem of securing trusted applications. It has reviewed and classified major security measures and investigated their limitations. Amongst others, it has noted that orthodox infrastructure access controls lack the facilities to guard trusted applications and argued that conventional intrusion detection systems tend to be vulnerable to desynchronisation. Viewed in terms of the distribution of effort, most contemporary approaches used to secure trusted applications can be seen to demand that application developers eliminate all flaws during construction or alternatively require that the producers of runtime security systems duplicate application logic and state faithfully. In the case of nontrivial applications these are substantial, arguably even unattainable, demands.

These issues motivate the thesis proposal for an interface between application and runtime security component, designed to allow the latter to monitor and adjust the former. The approach can be presented as a pragmatic compromise which makes it possible to share the security costs between application development and runtime security measures.

Viewed in terms of the problem of concentrating the security concerns of a computer system (described in Chapter 2), the contribution of the work consists of two primary parts:

- An analysis which argues that it currently does not appear feasible to transfer *completely* the security duties of some (*ie trusted*) applications to infrastructure components.

- A synthesis which proposes an interface designed to abstract and transfer a subset of the application security functions, namely the analysis phase, to an external component. While this strategy remains reliant on the support of the application provider to provide the information

and act on the analysis, it appears to be an attractive method of modulating application activity, whether to enforce classical policies or block suspected intrusions.

The work performed towards both the above parts can be presented as a sequence of interlocking tasks—the thesis:

- identified the challenge of extracting and managing the security functions of applications,
- formulated a model to describe this process,
- examined the limitations of several existing approaches with the aid of the model. The examination included both theoretical and practical aspects, the former presenting formalisms to describe the constraints, the latter contributing a NIDS desynchronisation attack involving variations in the handling of TCP urgent data,
- defined a class of applications, referred to as "*trusted*" whose security responsibilities are difficult to transfer to infrastructure systems,
- generated a taxonomy of security measures from the above model,
- used the taxonomy as a framework to position existing approaches and describe development trends,
- proposed a design which explores an approach involving a dedicated security interface between application developer and specialised security component,
- investigated several abstractions which can be used to structure the exchanges across the interface,
- constructed and released an operational implementation based on the design,
- exercised, optimised and evaluated the implementation
- and reflected on the utility of the approach.

The above are deemed to constitute an incremental, as opposed to a revolutionary, advance—the thesis has articulated an approach which assists in the control of security flaws, it has not discovered a means of eliminating them. Superficially this may appear less appealing, but it is consistent with

the compelling arguments of Brooks [26] that there are *"no silver bullets"* awaiting discovery which will solve the problem of constructing reliable and thus also secure software. Under these circumstances a system which can be used to manage the flaws in fielded systems does indeed have value.

Bibliography

[1] J. Abela, T. Debeaupuis, and E. Guttman. Universal format for logger messages. http://www.hsc.fr/gulp/, 1997.

[2] Serge Abiteboul. Querying semi-structured data. In *International Conference on Database Theory*, pages 1–18, January 1997.

[3] A. Acharya, Edjlali G., and Chaudhary V. History-based access control for mobile code. In *5th ACM Conference on Computer and Communications Security*, 1998.

[4] J. P. Anderson. Computer security technology planning study. Technical report, USAF Electronic Systems Division, October 1972.

[5] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., April 1980.

[6] R. Anderson. Why information security is hard - an economic perspective. In *17th Annual Computer Security Applications Conference*, December 2001.

[7] C. J. Antonelli, M. Undy, and P. Honeyman. The packet vault: Secure storage of network data. In *USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 103–110, April 1999.

[8] Apache Consortium. Apache HTTP server project. http://httpd.apache.org/, 1995.

[9] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *6th ACM Conference on Computer and Communications security*, pages 1–7, November 1999.

[10] S. Axelsson. Research in intrusion-detection systems: A survey. Technical report, Chalmers University of Technology, August 1999.

[11] S. Axelsson, U. Lindqvist, U. Gustafson, and E. Jonsson. An approach to unix security logging. In *21st National Information Systems Security Conference*, pages 62–75, October 1998.

[12] B. Bakker et al. log4cpp - log library for C++. http://sourceforge.net/projects/log4cpp, December 2000.

[13] A. Bali. IDS/A debian package. http://packages.debian.org/idsa/, June 2002.

[14] C. Beermann. Calamaris. http://calamaris.cord.de/, 1997.

[15] S. M. Bellovin. Computer security - an end state ? *Communications of the ACM*, 44(3):131–132, March 2001.

[16] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: hypertext transfer protocol - HTTP/1.0, May 1996.

[17] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: uniform resource identifiers (URI): Generic syntax, August 1998.

[18] M. Bishop. How to write a setuid program. *;login:*, 12(1):5–11, January 1987.

[19] M. Bishop. A model of security monitoring. In *5th Annual Computer Security Applications Conference*, pages 46–52, 1989.

[20] M. Bishop. A standard audit trail format. In *18th National Information Systems Security Conference*, pages 136–145, October 1995.

[21] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, May 1996.

[22] J. Biskup and U. Flegel. Transaction-based pseudonyms in audit data for privacy respecting intrusion detection. In *3rd International Symposium on Recent Advances in Intrusion Detection*, pages 28–48, October 2000.

[23] B. Blakley. The emperor's old armor. In *ACM New Security Paradigms Workshop*, pages 2–16, September 1996.

[24] E. Boebert. Some thoughts on the occasion of the NSA linux release. *Linux Journal*, January 2001.

[25] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[26] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[27] J. Bull, L. Gong, and K. Sollins. Towards security in an open systems federation. In *European Symposium on Research in Computer Security*, pages 3–20, November 1992.

[28] C. A. Carver and U. W. Pooch. An intrusion response taxonomy and its role in automatic intrusion response. In *IEEE Workshop on Information Assurance and Security*, pages 129–135, June 2000.

[29] J. Case, M. Fedor, M. Schoffstall, and J. Davin. RFC 1157: a simple network management protocol, May 1990.

[30] S. Castano, G. Martella, and P. Samarati. A new approach to security system development. In *ACM New Security Paradigms Workshop*, pages 82–88, August 1994.

[31] National Computer Security Center. US department of defense trusted computer system evaluation criteria, December 1985.

[32] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[33] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, April 2001.

[34] C. Chung, M. Gertz, and K. Levitt. DEMIDS: A misuse detection system for database systems. In *3rd Annual IFIP TC-11 WG 11.5 Working Conference on Integrity and Control in Information Systems*, 1999.

[35] The MITRE Corporation. Common vulnerabilities and exposures 20010918. http://cve.mitre.org/, September 2001.

[36] C. Cowan, C. Pu, and H. Hinton. Death, taxes and imperfect software: Surviving the inevitable. In *ACM New Security Paradigms Workshop*, pages 54–70, September 1998.

[37] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, pages 63–78, January 1998.

[38] T. de Raadt et al. Openbsd. http://www.openbsd.org/, 1997.

[39] G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. Web services security policy language (WS-SecurityPolicy), December 2002.

[40] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[41] Computer Economics. E flash. http://www.computereconomics.com/, January 2002.

[42] S. Elbaum and J. C. Munson. Intrusion detection through dynamic software measurement. In *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 41–50, April 1999.

[43] M. Erlinger, S. Staniford-Chen, et al. IETF intrusion detection working group. http://www.ietf.org/html.charters/idwg-charter.html, 1999.

[44] C. Fiebig. Patching is the problem, says Microsoft: Interview with Ian Thompson, May 2003.

[45] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: hypertext transfer protocol - HTTP/1.1, June 1999.

[46] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[47] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, May 1999.

[48] S. M. Furnell, G. B. Magklaras, M. Papadaki, and P. S. Dowload. A generic taxonomy for intrusion specification and response. In *Euromedia 2001*, April 2001.

[49] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, April 1999.

[50] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. RFC 2518: HTTP extensions for distributed authoring — WEBDAV, February 1999.

[51] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, pages 1–13, July 1996.

[52] G. S. Goldszmidt. *Distributed Management by Delegation.* PhD thesis, Columbia University, 1996.

[53] J. Grossman. TRACE used to increase the dangerous of XSS. Bugtraq Thread, http://www.securityfocus.com/, January 2003.

[54] P. Hallam-Baker, E. Maler, et al. Security assertion markup language (SAML), November 2002.

[55] L. R. Halme and R. K. Bauer. AINT misbehaving: A taxonomy of anti-intrusion techniques. In *18th National Information Systems Security Conference*, pages 163–172, October 1995.

[56] S. E. Hansen and E. T. Atkins. Centralized system monitoring with swatch. In *3rd USENIX Security Symposium*, pages 105–117, September 1992.

[57] S. E. Hansen and E. T Atkins. Automated system monitoring and notification with swatch. In *7th USENIX Systems Administration Conference*, November 1993.

[58] D. Hardeman. Ample. http://ample.sf.net/, July 2001.

[59] H. Hazewinkel, C. Kalbfleisch, and J. Schoenwaelder. RFC 2594: definitions of managed objects for www services, May 1999.

[60] L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *IEEE Symposium on Research in Security and Privacy*, pages 296–304, May 1990.

[61] S. A. Hofmeyr. *An Immunological Model of Distributed Detection and Its Application to Computer Security.* PhD thesis, University of New Mexico, May 1999.

[62] J. D. Howard. *An Analysis Of Security Incidents On The Internet.* PhD thesis, Carnegie Mellon University, April 1997.

[63] K. Ilgun. USTAT: A real-time intrusion detection system for unix. Master's thesis, University of California, July 1992.

[64] C. E. Irvine and D. Volpano. A practical tool for developing trusted applications. In *11th Annual Computer Security Applications Conference*, pages 190–195, December 1995.

[65] A. K. Jones and Y. Lin. Application intrusion detection using language library calls. In *17th Annual Computer Security Applications Conference*, December 2001.

[66] D. Kahneman, P. Slovic, and A. Tversky, editors. *Judgment under uncertainty: Heuristics and biases.* Cambridge University Press, 1982.

[67] I. Khalifa and M. Stahre. Teapop - a POP3 server daemon. http://www.toontown.org/teapop/, 1999.

[68] S. Kille and N. Freed. RFC 2789: mail monitoring MIB, March 2000.

[69] G. H. Kim and E. H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. In *3rd Annual System Administration, Networking and Security Conference*, pages 89–101, April 1994.

[70] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, August 2002.

[71] M. Kirkwood and I. Lynagh. Firewall kit. http://hairy.beasts.org/fk/, 2000.

[72] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *10th Annual Computer Security Applications Conference*, pages 134–144, 1994.

[73] M. Koster. A standard for robot exclusion. http://info.webcrawler.com/mak/projects/robots/norobots.html, June 1994.

[74] I. V. Krsul. *Software Vulnerability Analysis.* PhD thesis, Purdue University, May 1998.

[75] S. Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue University, August 1995.

[76] B. W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, pages 437–443, March 1971.

[77] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[78] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.

[79] W. Lee, S. J. Stolfo, and K. Mok. Mining audit data to build intrusion detection models. In *International Conference on Knowledge Discovery and Data Mining*, September 1998.

[80] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, April 1999.

[81] U. Lindqvist and E. Jonsson. How to systematically classify computer security intrusions. In *IEEE Symposium on Security and Privacy*, May 1997.

[82] U. Lindqvist and E. Jonsson. A map of security risks associated with using cots. *Computer*, 31(6):60–66, June 1998.

[83] H. F. Lipson and D. A. Fisher. Survivability — a new technical and business perspective on security. In *ACM New Security Paradigms Workshop*, pages 33–39, September 1999.

[84] B. Long et al. The common gateway interface 1.1. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html, November 1995.

[85] C. Lonvick. RFC 3164: the BSD syslog protocol, August 2001.

[86] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*, pages 303–314, October 1998.

[87] C. Lott. csize - measure the size of c source files. http://www.chris-lott.org/resources/software/, September 1994.

[88] T. F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *11th National Computer Security Conference*, October 1988.

[89] A. Luotonen. The common logfile format. http://www.w3.org/pub/WWW/Daemon/User/Config/Logging.html, 1995.

[90] T. J. Mc Cabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[91] C. Meadows. An outline of a taxonomy of computer security research and development. In *ACM New Security Paradigms Workshop*, pages 33–35, August 1993.

[92] C. C. Michael and A. Ghosh. Two state-based approaches to program-based anomaly detection. In *16th Annual Computer Security Applications Conference*, December 2000.

[93] D. Mosberger and T. Jin. httperf — a tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[94] I. S. Moskowitz and M. H. Kang. An insecurity flow model. In *ACM New Security Paradigms Workshop*, pages 61–74, September 1997.

[95] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Universite de Namur, September 1997.

[96] J. Myers and M. Rose. RFC 1939: post office protocol - version 3, May 1996.

[97] T. Olovsson. A structured approach to computer security. Technical report, Chalmers University of Technology, 1992.

[98] V. Paxson. Bro: A system for detecting network intruders in real-time. In *7th USENIX Security Symposium*, pages 31–50, January 1998.

[99] M. Petkac and L. Badger. Security agility in response to intrusion detection. In *16th Annual Computer Security Applications Conference*, December 2000.

[100] R. Pike. System software research is irrelevant. Talk Slides, February 2000.

[101] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The common intrusion detection framework architecture. http://www.gidos.org/, 2000.

[102] J. Postel. RFC 791: darpa internet program protocol specification, September 1981.

[103] T. E. Potok, M. Vouk, and A. Rindos. Productivity analysis of object-oriented software development in a commercial environment. *Software — Practice and Experience*, 29(10):833–847, 1999.

[104] T. H. Ptacek and T. N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, 1998.

[105] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity in operating systems. In *ICMAS Workshop on Immunity-Based System*, 1996.

[106] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *5th USENIX Symposium on Operating System Design and Implementation*, December 2002.

[107] Rain Forrest Puppy. Whisker. http://www.wiretrip.net/rfp/, 1999.

[108] M. J. Ranum and P. D. Robertson. Logging data attribute map. http://www.ranum.com/logging/logging-data-map.html, August 2002.

[109] M. K. Ranum and F. M. Avolio. A toolkit and methods for internet firewalls. In *Proceedings of the USENIX Conference*, pages 37–44, 1994.

[110] E. S. Raymond. Fetchmail - a remote-mail retrieval and forwarding utility. http://www.tuxedo.org/

[111] J. Riordan and D. Alessandri. Target naming and service apoptosis. In *3rd International Symposium on Recent Advances in Intrusion Detection*, pages 217–225, October 2000.

[112] M. Roesch. Snort — lightweight intrusion detection for networks. In *13th USENIX Systems Administration Conference*, November 1999.

[113] T. Rooker. Application level security using an object-oriented graphical user interface. In *ACM New Security Paradigms Workshop*, pages 105–108, August 1993.

[114] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Procedings of the IEEE*, 63(9):1278–1308, September 1975.

[115] V. Samar and R. Schemers. Unified login with pluggable authentication modules (PAM), October 1995.

[116] R. S. Sandhu. The typed access matrix model. In *IEEE Symposium on Security and Privacy*, pages 122–136, May 1992.

[117] M. Schaefer. The new security paradigms workshop - boom or bust ? panel position statement. In *ACM New Security Paradigms Workshop*, pages 119–123, September 2001.

[118] E. A. Schneider. Security architecture-based system design. In *ACM New Security Paradigms Workshop*, pages 25–31, September 1999.

[119] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Dagstuhl 10th Anniversary Symposium*, August 2000.

[120] B. Schneier. Managed security monitoring: Closing the window of exposure. Technical report, Counterpane Internet Security, 2000.

[121] R. Sekar, T. Bowen, and Segal. M. On preventing intrusions by process behavior monitoring. In *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 29–40, April 1999.

[122] R. S. Sielken. Application intrusion detection. Master's thesis, University of Virginia, May 1999.

[123] Secure Software. RATS: Rough auditing tool for security. http://www.securesoftware.com/rats/, May 2001.

[124] European Community Advisory Group SOG-IS. Information technology evaluation security criteria, June 1991.

[125] Solar Designer. Non-executable user stack. http://www.openwall.com/, 1997.

[126] A. Somayaji and S. Forrest. Automated response using system-call delays. In *9th USENIX Security Symposium*, August 2000.

[127] D. Song. Fragrouter. http://www.anzen.com/research/nidsbench/, 1999.

[128] S. Staniford-Chen, B. Tung, and D. Schnackenberg. The common intrusion detection framework (CIDF). In *Information Survivability Workshop*, October 1998.

[129] G. Stocksdale. NSA glossary of terms used in security and intrusion detection. http://www.sans.org/newlook/resources/glossary.htm, April 1998.

[130] T. Sullivan. All things web. http://www.pantos.org/atw/, 1999.

[131] Sun Microsystems. *SunSHIELD Basic Security Module Guide*, November 1995.

[132] M. Swimmer. Review and outlook of the detection of viruses using intrusion detection systems. http://www.raid-symposium.org/raid2000/, October 2000.

[133] The Open Group. Distributed audit service (XDAS) base. http://www.opengroup.org/pubs/, 1997.

[134] A. Tversky and D. Kahneman. The framing of decisions and the psychology of choice. *Science*, 211(4481):453–458, January 1981.

[135] W. Venema. TCP wrapper, network monitoring, access control and booby traps. In *3rd USENIX Security Symposium*, pages 85–92, September 1992.

[136] I. Welch and R. Stroud. Reflection as a mechanism for enforcing security policies in mobile code. In *6th European Symposium on Research in Computer Security*, October 2000.

[137] M. Welz and A. Hutchison. Incremental security in open, untrusted networks. In *Future Trends in Distributed Computer Systems*, pages 151–154, November 1999.

[138] M. Welz and A. Hutchison. Interfacing trusted applications with intrusion detection systems. In *4th International Symposium on Recent Advances in Intrusion Detection*, pages 37–53, October 2001.

[139] D. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/, May 2001.

[140] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, 2002.

[141] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *8th USENIX Security Symposium*, pages 79–91, 1999.

[142] D. Zamboni. *Using Internal Sensors for Computer Intrusion Detection.* PhD thesis, Purdue University, August 2001.

[143] M. Zelem, M. Pikula, and M. Ockajak. Medusa DS9 security system. http://medusa.fornax.sk/, 1999.

[144] D. Zerkle and K. Levitt. Netkuang — a multi-host configuration vulnerability checker. In *6th USENIX Security Symposium*, pages 195–201, July 1996.

Appendix A

Manual Pages

The appendix contains the manual pages referenced in the previous chapters. Included are the pages describing the server `idsad`, parts of the C API, a number of predefined abstractions as well as the documentation index.

The manual pages describing the configuration file format, the different types, the various support utilities, analysis and response modules, wrappers and replacements have been omitted. However, they can be found in source archives `idsa-0.96.0.tar.gz` or newer.

IDSA(1) IDSA(1)

NAME

idsa – idsa system overview

SYNOPSIS

idsa is a combined intrusion detection, access control and logging system for applications. Its documentation is distributed over several manual pages.

ADMINISTRATION

The following pages describe the runtime:

idsad(8)	Master daemon
idsad.conf(5)	Master configuration file
idsaexec(8)	Utility to run commands from **idsad**(8)
idsapipe(8)	Utility to relay events from **idsad**(8)
idsaguile(8)	Interface to **guile**(1)
idsapid(8)	RC support for master daemon
idsaguardgtk (8)	GTK utility for interactive control
idsaguardtty(8)	TTY utility for interactive control

Documentation for selected modules loaded by **idsad**(8) to perform analysis and response tasks:

mod_chain(8)	Identify a rule chain
mod_constrain(8)	String anomaly detector
mod_counter(8)	Set and test counters
mod_exists(8)	Test if a field exists
mod_interactive(8)	Interface to **idsaguardgtk**(8) or **idsaguardtty**(8)
mod_keep(8)	Retain state
mod_length(8)	Measure field sizes
mod_log(8)	Write to file or subprocess
mod_regex(8)	Match regular expression
mod_send(8)	Return extra data to application
mod_time(8)	Match a given time
mod_timer(8)	Match a given period
mod_true(8)	Always match
mod_truncated(8)	Test for incomplete fields
mod_type(8)	Determine field type

DEVELOPMENT

The following pages document the API used to add **idsa** support to applications:

Higher Level C API:

idsa_open(3)	Initialise interface
idsa_close(3)	Terminate interface
idsa_set(3)	Report an event
idsa_types(4)	Type information

Shell interface:

idsalog(1)	Report and request permission to perform an action

IDS/A System March 2003 1

IDSA(1) IDSA(1)

ABSTRACTIONS

Frameworks to report activity in a form less dependent on the application domain:

idsa-scheme-am(7)	Access control abstraction
idsa-scheme-clf(7)	HTTP common logging format
idsa-scheme-err(7)	Error reporting abstraction
idsa-scheme-fnl(7)	Functionality labelling
idsa-scheme-ldm(7)	Logging data map
idsa-scheme-rq(7)	Resource control
idsa-scheme-ssm(7)	State management

MISCELLANEOUS

Replacements and utilities:

idsaklogd(8)	Kernel logger replacement
idsarlogd(8)	Remote syslog replacement
idsasyslogd(8)	Local syslog replacement
idsatcpd(8)	TCP wrapper replacement
idsatcplogd(8)	TCP connection logger

AUTHOR

Marc Welz

COPYING

idsa may only be distributed and modified in accordance with the terms of the **GPL (GNU General Public License)** as published by the **FSF**(Free Software Foundation).

SEE ALSO

idsad(3), **idsad.conf**(5).

IDSAD(8) IDSAD(8)

NAME

idsad – master daemon of the idsa system

SYNOPSIS

idsad [**-hknuv**] [**-f** *file*] [**-i** *username*] [**-M** *integer*] [**-p** *socket ...*] [**-r** *directory*]

DESCRIPTION

idsad accepts connections from one or more unix domain sockets. Security events are read from established connections, processed according to a rule set defined in **idsad.conf**(5) and the replies returned.

OPTIONS

-f file Read configuration from *file* instead of default */etc/idsad.conf*

-h Print a terse help message

-i username
Run as specified user instead of root. Group is determined from password file entry

-k Kill and replace an already running **idsad** instance. Note that this replacement is atomic, at no point in time will the listening sockets be unbound. This option should be used to reconfigure **idsad** in place of sending it a **SIGHUP**. Reconfiguring or restarting **idsad** via a signal is not feasible as the running **idsad** instance has no way of recovering the privileges it relinquished on entering chrooted environment and changing its user identifier

-M integer
Preallocate memory resources to support the given number of clients. In this configuration **idsad** will allocate **no** memory after initialization --- this should prevent memory exhaustion failures. Using a value exceeding the number of available file descriptors is inadvisable, as this will disable the mechanism which enforces connection quotas on non-root users

-n Do not fork into background

-p socket ...
Listen on one or more unix domain sockets. Multiple sockets should be separated by unquoted whitespaces. If this option is omitted, **idsad** will default to listening on */var/run/idsa*

-r directory
Chroot to *directory* after initialisation

-u Honour umask when creating sockets. Allows the system administrator to restrict access to the socket to a given group or user. Note that few systems other than Linux honour permission bits for unix domain sockets

-v Print version number

FILES

/etc/idsad.conf
Default **idsad** configuration file.

/var/run/idsa
Default listen socket.

AUTHOR

Marc Welz

COPYING

idsa may only be distributed and modified in accordance with the terms of the **GPL (GNU General Public License)** as published by the **FSF**(Free Software Foundation).

SEE ALSO

idsad.conf(5), **syslogd**(8).

IDS/A System JULY 2000 1

IDSA_OPEN(3) IDSA_OPEN(3)

NAME

idsa_open – create an idsa handle

SYNOPSIS

#include <idsa.h>

IDSA_CONNECTION *idsa_open(char *_name_, char *_credential_, int _flags_);

DESCRIPTION

idsa_open allocates a handle which is used in subsequent calls to **idsa_set**, **idsa_scan** and others. *name* is the name used to identify the client application, *credential* a token used for authentication (currently unused, should be **NULL**). *flags* can be used to set zero or more of the following options, bitwise or'd:

IDSA_F_FAILOPEN
Change the behaviour of **idsa_log**, **idsa_set** and **idsa_scan** to return **IDSA_L_ALLOW** instead of **IDSA_L_DENY** in case of failures internal to idsa.

IDSA_F_ENV
allow a user to override the location of the **idsa** socket using **IDSA_SOCKET**. The use of this is inadvisable for setuid programs.

IDSA_F_SIGPIPE
Do not trap **SIG_PIPE**. This only applies to older platforms, newer ones do require that that any signals be trapped.

IDSA_F_UPLOAD
Enable the uploading of rules (and thus shared objects) into the process space. The use of this option is inadvisable for applications which are more trusted than **idsad**(8).

IDSA_F_NOBACKOFF
Disable the linear backoff strategy in case of a failure of **idsad**(8).

IDSA_F_TIMEOUT
Set a timeout for I/O to **idsad**(8). Using this flag in conjunction with **IDSA_F_UPLOAD** is not advised.

IDSA_F_KEEP
Disable automatic deallocation of events inside **idsa_log**.

RETURN VALUE

A pointer to an **IDSA_CONNECTION** structure on success, **NULL** otherwise.

FILES

/etc/idsad.conf
Default **idsad**(8) configuration file.

/var/run/idsa
Default listen socket.

AUTHOR

Marc Welz

COPYING

idsa may only be distributed and modified in accordance with the terms of the **GPL (GNU General Public License)** as published by the **FSF (Free Software Foundation)**.

SEE ALSO

idsa_close(3).

IDS/A System JULY 2001 1

IDSA_CLOSE(3) IDSA_CLOSE(3)

NAME

idsa_close – destroy an idsa handle

SYNOPSIS

#include <idsa.h>

int idsa_close(IDSA_CONNECTION *c);

DESCRIPTION

idsa_close deallocates the resources associated with the given **idsa** connection handle.

RETURN VALUE

Zero on success, nonzero otherwise.

FILES

/etc/idsad.conf
Default **idsad**(8) configuration file.

/var/run/idsa
Default listen socket.

AUTHOR

Marc Welz

COPYING

idsa may only be distributed and modified in accordance with the terms of the **GPL (GNU General Public License)** as published by the **FSF (Free Software Foundation)**.

SEE ALSO

idsa_open(3).

IDSA_SET(3) IDSA_SET(3)

NAME

idsa_set, idsa_scan – report an event to the idsa system

SYNOPSIS

#include <idsa.h>

int idsa_set(IDSA_CONNECTION *c, char *n, char *s, int f, unsigned ar, unsigned cr, unsigned ir, ...);

int idsa_scan(IDSA_CONNECTION *c, char *n, char *s, int f, unsigned ar, unsigned cr, unsigned ir, ...);

DESCRIPTION

idsa_set and **idsa_scan** create an event from their parameters. This event is then reported to the idsa system via the connection handle *c*.

The event name is given in *n* and its scheme or namespace in *s*. The combination of *n* and *s* should identify an event uniquely.

f is a flag which indicates whether the application can and will honour a request to deny this event. If *f* is zero the application won't honour an **IDSA_L_DENY**, otherwise it will.

ar, *cr* and *ir* contain the risks to availability, confidentiality and integrity associated with permitting the event. Risks are local to an application namespace. Risks can be specified as pair of cost, confidence values using **idsa_risk_make** or the following predefined risks can be used:

IDSA_R_TOTAL	1.000	0.990	Complete failure
IDSA_R_PARTIAL	0.500	0.750	Partial failure
IDSA_R_MINOR	0.250	0.875	Minor failure
IDSA_R_NONE	0.000	0.990	No significant risk
IDSA_R_UNKNOWN	0.000	0.000	Unknown risk

idsa_set and **idsa_scan** only differ in the way optional arguments are handled. Optional arguments are given as a triple of *name*, *type* and *value* where *name* is a character string, *type* an unsigned integer from the set of available types. For **idsa_set** the value is a pointer to the in-memory representation of the value while for **idsa_scan** the value is a character string. The last argument passed to both functions has to be **NULL**.

EXAMPLES

```
pid_t target;
char *name;
idsa_set(c, "kill", "audited-shell", 1,
  IDSA_R_PARTIAL, IDSA_R_NONE, IDSA_R_UNKNOWN,
  "victim-pid", IDSA_T_PID, &target,
  "victim-name", IDSA_T_STRING, name,
  NULL);
```

Request permission (*f* set to **1**) to terminate the process identified by *target*. The event poses an elevated risk to availability, little risk to confidentiality and an unknown threat to integrity.

```
char *target;
char *name;
idsa_scan(c, "kill", "audited-shell", 1,
  IDSA_R_PARTIAL, idsa_risk_make(0.0, 0.99), IDSA_R_UNKNOWN,
  "victim-pid", IDSA_T_PID, target,
  "victim-name", IDSA_T_STRING, name,
  NULL);
```

This example has the same effect as the previous one, but involves **idsa_scan**, not **idsa_set**. Consequently the process id in the variable *target* is represented as a string to be parsed by **idsa_set**. Also note that the risk to confidentiality is specified directly using **idsa_risk_make**.

```
struct sockaddr_in sa;
char *method, *url;
idsa_set(c, "http-request", "samplehttpd", 1,
  IDSA_R_UNKNOWN, IDSA_R_PARTIAL, IDSA_R_UNKNOWN,
  "client-ip", IDSA_T_SADDR, &sa,
  "http-method", IDSA_T_STRING, method,
  "url", IDSA_T_STRING, url,
  NULL);
```

This example shows a client request as reported by the web server. The optional fields report the IP address of the client, the method and the requested url. The event poses a partial threat to confidentiality, while the risks to availability and integrity are not rated.

RETURN VALUE

IDSA_L_ALLOW if the event is permitted, **IDSA_L_DENY** if it is to be denied. By default an internal failure will generate an **IDSA_L_DENY** return value, but this can be changed with the **IDSA_F_FAILOPEN** flag to **idsa_open**.

FILES

/etc/idsad.conf
Default **idsad**(8) configuration file.

/var/run/idsa
Default listen socket.

AUTHOR

Marc Welz

COPYING

idsa may only be distributed and modified in accordance with the terms of the **GPL (GNU General Public License)** as published by the **FSF (Free Software Foundation)**.

SEE ALSO

idsa_open(3), **idsa_close**(3), **idsa_types**(3).

IDSA-SCHEME-AM(7) IDSA-SCHEME-AM(7)

NAME

IDS/A Access Control Abstraction

DESCRIPTION

This page lists the event field names which should be passed to **idsa_set**(3) or **idsa_scan**(3) to describe security events in terms of an access control model. The primary fields are **IDSA_AM_SUBJECT**, **IDSA_AM_OBJECT**, and **IDSA_AM_ACTION**, with several types of access actions or modes defined. Additional fields are used to report compartments, levels and roles.

FIELDS

IDSA_AM_SUBJECT
: Active entity which attempts the action

IDSA_AM_OBJECT
: Passive entity on which the action is performed

IDSA_AM_ACTION
: Type of action performed. The following string values are defined:

  IDSA_AM_AREAD
  : Transfer information to the subject

  IDSA_AM_AWRITE
  : Transfer information from the subject

  IDSA_AM_ACREATE
  : Generate object

  IDSA_AM_ADESTROY
  : Render object inaccessible

  IDSA_AM_AFLOW
  : Untrusted transfer: Subject gains sufficient control of object so that subsequent actions performed by object could have been performed at the request of this subject. Used to measure insecurity flow. Appropriate for invoking nonsetuid executables, changing uid, remote logins, etc

  IDSA_AM_AREQUEST
  : Trusted transfer: Subject requests a trusted object to perform an action. Appropriate for execution of setuid executables, protocol commands

  IDSA_AM_AOTHER
  : Fallback if none of the above are appropriate

IDSA_AM_CLEARANCE
: Security level of subject

IDSA_AM_CLASSIFICATION
: Security level of object

IDSA_AM_DOMAIN
: Compartment of subject

IDSA_AM_TYPE
Compartment of object

IDSA_AM_ROLE
Role assumed by subject

EXAMPLE

```
struct sockaddr_in sa;
char *url;
 ...
idsa_set(c, "reply", "samplehttpd", 1,
   IDSA_R_UNKNOWN, IDSA_R_PARTIAL, IDSA_R_MINOR,
   IDSA_AM_SUBJECT, IDSA_T_SADDR, &sa,
   IDSA_AM_OBJECT, IDSA_T_STRING, url,
   IDSA_AM_ACTION, IDSA_T_STRING, IDSA_AM_READ,
   ...
   NULL);
```

Request permission to allow a client access retrieve an url.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).

IDSA-SCHEME-CLF(7) IDSA-SCHEME-CLF(7)

NAME

IDS/A Common Logfile Format Mapping

DESCRIPTION

This page lists the event field names which should be passed to **idsa_set**(3) or **idsa_scan**(3) to describe security events using the Common Logfile Format (**CLF**) as documented by Luotonen *et al* in *http://www.w3.org/pub/WWW/Daemon/User/Config/Logging.html*.

FIELDS

IDSA_CLF_REMOTEHOST
: Remote hostname (or IP number if no DNS hostname is available or name resolution has been disabled)

IDSA_CLF_RFC931
: The remote logname of the user

IDSA_CLF_AUTHUSER
: The username as which the user has authenticated himself

IDSA_CLF_DATE
: Date and time of the request. *Note* that **idsa** events also provide a timestamp

IDSA_CLF_REQUEST
: The request line exactly as it came from the client

IDSA_CLF_STATUS
: The **HTTP** status code returned to the client

IDSA_CLF_BYTES
: The content-length of the document transferred

EXAMPLE

```
struct sockaddr_in sa;
int bytes, code;
char *req;
 ...
code = 404;
 ...
idsa_set(c, "reply", "samplehttpd", 0,
   IDSA_R_MINOR, IDSA_R_NONE, IDSA_R_NONE,
   IDSA_CLF_REMOTEHOST, IDSA_T_SADDR, &sa,
   IDSA_CLF_STATUS,  IDSA_T_INT, &code,
   IDSA_CLF_BYTES,   IDSA_T_INT, &bytes,
   IDSA_CLF_REQUEST, IDSA_T_STRING, req,
   ...
   NULL);
```

Report a web page access.

IDS/A System APRIL 2003 1

SEE ALSO

idsa_set(3), idsa_scan(3), idsad(8).

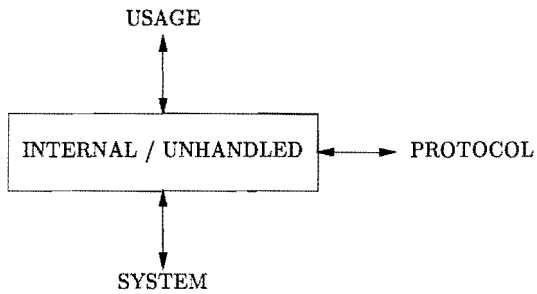
IDSA-SCHEME-ERR(7) IDSA-SCHEME-ERR(7)

NAME

IDS/A Error Reporting Scheme

DESCRIPTION

This page documents an abstraction which describes errors and their location. The abstraction considers the entity reporting the errors to be part of a layered design, where errors may occur above, inside, adjacent and below the reporting system.



Supplementary fields **IDSA_ES_SYS_ERRNO**, **IDSA_ES_SYS_EXIT** and **IDSA_ES_PRO_HTTP** provide more detailed information for particular subsystems.

FIELDS

IDSA_ES

The type of error reported. It should have one of the following value strings:

IDSA_ES_USAGE
: Failure at user interface (layer above reporting system)

IDSA_ES_SYSTEM
: Failure at infrastructure (layer below reporting system)

IDSA_ES_PROTOCOL
: Failure at logical peer (system at same layer, external to reporting system)

IDSA_ES_UNHANDLED
: Anticipated but unhandled condition internal to reporting system

IDSA_ES_INTERNAL
: Unanticipated failure internal to reporting system

IDSA_ES_OTHER
: Other or unknown type of error

IDSA_ES_SYS_ERRNO
: Unix error number associated with error, listed in */usr/include/errno.h*

IDSA_ES_SYS_EXIT
: Unix exit code associated with error, listed in */usr/include/sysexists.h*

IDSA_ES_PRO_HTTP
: RFC2068 HTTP error status codes

IDS/A System APRIL 2003 1

EXAMPLE

```
struct sockaddr_in sa;
int status;
 ...
status = 401; /* not authorised */
 ...
idsa_set(c, "not-found", "samplehttpd", 0,
    IDSA_R_MINOR, IDSA_R_NONE, IDSA_R_NONE,
    "client-ip", IDSA_T_SADDR, &sa,
    IDSA_ES, IDSA_T_STRING, IDSA_ES_PROTOCOL,
    IDSA_ES_PRO_HTTP, IDSA_T_INT, &status,
    ...
    NULL);
```

Report a request for an entity which the peer is not permitted to access.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).

IDSA-SCHEME-FNL(7) IDSA-SCHEME-FNL(7)

NAME

IDS/A Functionality Scheme

DESCRIPTION

This page describes a simple, discrete functionality labelling abstraction. A similar, continuous approximation can be implemented using the required **arisk** field, with important functions having a high cost to availabilty if denied.

FIELDS

IDSA_FNL_LEVEL

The functionality level of the current event, with essential operations starting at 0

IDSA_FNL_MAX

The highest functionality level

EXAMPLES

```
struct sockaddr_in sa;
 ...
idsa_set(c, "http-request", "samplehttpd", 1,
   ...
   "client-ip", IDSA_T_SADDR, &sa,
   "http-method", IDSA_T_STRING, "GET",
   IDSA_FNL, IDSA_T_INT, 0,
   NULL);
 ...
idsa_set(c, "http-request", "samplehttpd", 1,
   ...
   "client-ip", IDSA_T_SADDR, &sa,
   "http-method", IDSA_T_STRING, "MKCOL",
   IDSA_FNL, IDSA_T_INT, 3,
   NULL);
```

Report a basic function as well as a less frequently used extension.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).

IDSA-SCHEME-LDM(7) IDSA-SCHEME-LDM(7)

NAME

IDS/A Logging Data Map Mapping

DESCRIPTION

This page lists the event field names which should be passed to **idsa_set**(3) or **idsa_scan**(3) to describe security events using the Logging Data Map (**LDM**) proposed by Ranum and Robertson in *http://www.ranum.com/logging/logging-data-map.html*. The explanations are reproduced largely verbatim.

FIELDS

IDSA_LDM_NDATE
: Normal Date/Time Normalized (ISO8601) Date/Time format. *Note* that **idsa** also provides a timestamp

IDSA_LDM_SOURCEID
: Source host identifier

IDSA_LDM_TRANSID
: Transaction or message identifier

IDSA_LDM_PRIO
: Snort priority rating in the range 0-11:

0	Not suspicious traffic
1	Unknown traffic
2	Potentially bad traffic
3	Attempted information leak
4	Information leak
5	Large scale information leak
6	Attempted denial of service
7	Denial of service attack
8	Attempted user privilege gain
9	User privilege gain
10	Attempted administrator privilege gain
11	Administrator privilege gain

IDSA_LDM_REFS
: Event-IDs of related records, comma separated in form of message-ID@Source-ID

IDSA_LDM_GEOLOC
: Geographic location if known or relevant (arbitrary format)

IDSA_LDM_GROUP
: Grouping for administrative purposes (Arbitrary site private) (e.g.: "sales", "west coast", ...)

IDSA_LDM_RAWMSG
: Original message text, raw evidence form

IDSA_LDM_DESCRIPT
: Short description of event (human readable) ("failed login attempt" "message delivery notification" ...)

IDSA_LDM_OPERATION
: Description of operation performed (arbitrary string: "get file", POST, stat=sent, exec)

IDSA_LDM_PROTO
: Protocol in use for event (IPV6, TCP, UDP, ICMP, HTTP, HTTPS, FTP, ...)

IDSA_LDM_ALERTMSG
: Short description of alert (human readable) if appropriate ("Denial of service attack deflected..")

IDSA_LDM_ERRMSG
: Error message associated with the event, if any

IDSA_LDM_SRCPID
: Source process ID (if appropriate) of related process

IDSA_LDM_SRCIDENT
: Source of record (kernel, application, device, app name, or proc name)

IDSA_LDM_SRCUSER
: User-ID or name of attributed user

IDSA_LDM_TARGUSER
: User-ID or name of target/victim/destination user if applicable

IDSA_LDM_SRCDEV
: Source device or host platform (hostname, ip, mac address) identifier

IDSA_LDM_TARGDEV
: Target device or host platform (hostname, ip, mac address) identifier

IDSA_LDM_SRCCRED
: Credential presented by source user if any (password, password text, crypto key, cookie)

IDSA_LDM_TARGCRED
: Credential presented for use at destination if any (password, password text, crypto key, cookie)

IDSA_LDM_SRCPATH
: Source Pathname (URI, filename, executable to run, ...) (Windows pathnames should include Device: specifier if available)

IDSA_LDM_TARGPATH
: Target Pathname (URI, filename, executable to run, ...)

EXAMPLE

```
struct sockaddr_in sa;
char *url;
...
idsa_set(c, "reply", "samplehttpd", 0,
    IDSA_R_MINOR, IDSA_R_NONE, IDSA_R_NONE,
    IDSA_LDM_SOURCEID, IDSA_T_SADDR, &sa,
    IDSA_LDM_PROTO,    IDSA_T_STRING, "HTTP",
    IDSA_LDM_TARGPATH, IDSA_T_STRING, url,
    ...
    NULL);
```

Report a web page access.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).

IDSA-SCHEME-RQ(7) IDSA-SCHEME-RQ(7)

NAME

IDS/A Resource Accounting Abstraction

DESCRIPTION

This page lists the event field names which should be passed to **idsa_set**(3) or **idsa_scan**(3) to describe the resources held by a monitored application. Each resource is considered to consist of a number of discrete elements which are either allocated/consumed or available for allocation/consumption. An application in its simplest form would report the resources as a count, using either **IDSA_RQ_REQUEST**, **IDSA_RQ_RELEASE** or **IDSA_RQ_USED** as well the field **IDSA_RQ_UNITS** to describe the entities counted. More elaborate systems may also list totals, report unallocated elements and categorise resources.

FIELDS

IDSA_RQ_REQUEST
: The resources which are requested by the application emitting the security event

IDSA_RQ_RELEASE
: The resources released by the application. This field differs from the inverse of the previous one in that the release of resources is not blocked

IDSA_RQ_USED
: The number of resource items already held by the monitored entity, not counting the ones currently being requested or about to be released

IDSA_RQ_FREE
: The number of resource elements which could potentially still be allocated to the requesting system

IDSA_RQ_TOTAL
: The overall number of resource elements

IDSA_RQ_UNITS
: The units in which the reported resource elements are measured. Loads can be expressed as a number of units per time interval.

IDSA_RQ_CLASS
: The resource type held or consumed. The abstraction proposes three major categories (which in turn could be divided further):

  IDSA_RQ_CTRAFFIC
  : Resources related to transfers or communications, such as bytes of network traffic or blocks of disk IO

  IDSA_RQ_CPROCESSOR
  : Resources devoted to manipulation or transformation, such as CPU time

  IDSA_RQ_CSTORAGE
  : Resources required for storage, such as disk blocks or bytes or RAM

EXAMPLE

```
int size;
...
```

IDS/A System APRIL 2003 1

```
idsa_set(c, "multimedia-reply", "samplehttpd", 1,
    ...
    "content-type, IDSA_T_STRING, "video/mpeg",
    IDSA_RQ_REQUEST, IDSA_T_INT, &size,
    IDSA_RQ_UNITS, IDSA_T_STRING, "bytes-sent",
    IDSA_RQ_CLASS, IDSA_T_STRING, IDSA_RQ_CTRAFFIC,
    NULL);
```

Request permission to transfer a video to a client of a given number of bytes.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).

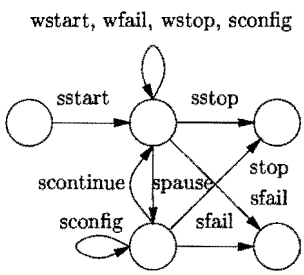
IDSA-SCHEME-SSM(7) IDSA-SCHEME-SSM(7)

NAME

IDS/A Service State Model

DESCRIPTION

This page presents a simple state model which represents the monitored system as a set of transitions. These are shown in the state diagram given below. Note that the depicted finite state automaton can be extended to a pushdown automaton where **wstart** pushes a symbol and both **wstop** and **wfail** consume it in order to encode the constraint that a unit of work has to be initiated before it can complete or fail.



FIELDS

IDSA_SSM

This field contains the transitions made by the application. It should have of the following value strings:

IDSA_SSM_SSTART
: Service ready to do work

IDSA_SSM_SPAUSE
: Service suspended

IDSA_SSM_SCONTINUE
: Service continued

IDSA_SSM_SCONFIG
: Service reconfigured

IDSA_SSM_SSTOP
: Service completed successfully

IDSA_SSM_SFAIL
: Service terminated abnormally

IDSA_SSM_WSTART
: Started a unit of work

IDSA_SSM_WSTOP
: Completed a unit of work successfully

IDSA_SSM_WFAIL
: Failed to complete a unit of work

EXAMPLES

```
struct sockaddr_in sa;
int fd, status, sz;
```

IDS/A System APRIL 2003 1

```
...
sz = sizeof(sa);
fd = accept(lfd, &sa, &sz);
idsa_set(c, "new-request", "samplehttpd", 1,
    ...
    "client-ip", IDSA_T_SADDR, &sa,
    IDSA_SSM, IDSA_T_STRING, IDSA_SSM_WSTART,
    NULL);
...
status = 404;
idsa_set(c, "new-request", "samplehttpd", 0,
    ...
    "http-status", IDSA_T_INT, &status,
    IDSA_SSM, IDSA_T_STRING, IDSA_SSM_WFAIL,
    NULL);
```

Report an HTTP request on arrival, as well as the failure to service it.

SEE ALSO

idsa_set(3), **idsa_scan**(3), **idsad**(8).